Harnessing Java™ 7

A Comprehensive Approach to Learning Java™

Volume - 3

Kishori Sharan

SAMPLE CHAPTERS

Network Programming and Java Native Interface (JNI)

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:

November 2011 First Print

ISBN-10: 1-46-633539-4

ISBN-13: 978-1-466-33539-4

Dedicated

To

My brothers Janki Sharan and Dr. Sita Sharan,
My wife Ellen Sharan, and
My Sister Ratna Kumari

Table of Contents

Preface ..... **i**

Structure of the Book ii
Audience iii
How to Use This Book iii
Java 7 New Features iv
Acknowledgements iv
Source Code and Errata vi
Questions and Comments vi

Chapter 3. Network Programming **7**

What is Network Programming? 7
IP Addressing Scheme 11
IPv4 Addressing Scheme 12
IPv6 and its Addressing Scheme 14
Special IP Addresses 15
Loopback IP Address 15
Unicast IP Address 16
Multicast IP Address 16
Anycast IP Address 17
Broadcast IP Address 17
Unspecified IP Address 18
Port Numbers 18
Socket API and Client-Server Paradigm 19
Representing a Machine Address 23
Representing a Socket Address 25
Creating a TCP Server Socket 26
Creating a TCP Client Socket 31
Putting a TCP Server and TCP Clients Together 33
Working with UDP Sockets 34
Creating a UDP Echo Server 37
A Connected UDP Socket 41
UDP Multicast Sockets 42
URI, URL and URN 44
URI and URL as Java Objects 48
Accessing the Contents of a URL 51
Non-Blocking Socket Programming 59
Socket Security Permissions 71

Asynchronous Socket Channels ... 72
Setting up an Asynchronous Server Socket Channel ... 74
Setting up an Asynchronous Client Socket Channel ... 80
Putting the Server and the Client Together ... 84
Datagram-Oriented Socket Channels ... 85
Multicasting Using Datagram Channels ... 91
Further Reading ... 95

Chapter 6. Java Native Interface ... 97

What is Java Native Interface? ... 97
System Requirements ... 98
Getting Started with JNI ... 98
Writing Java Program ... 98
Compiling the Java Program ... 101
Creating a C/C++ Header File ... 102
Writing the C/C++ Program ... 103
Creating a Shared Library ... 104
Creating a Shared Library Using Visual Studio 2005 ... 104
Running the Java Program ... 108
Native Function Naming Rules ... 109
Data Type Mapping ... 111
Using JNI Functions in C/C++ ... 113
Working with Strings ... 113
Working with Arrays ... 116
Accessing Java Objects in Native Code ... 120
Getting a Class Reference ... 120
Accessing Fields and Methods of the Java Object/Class ... 121
Creating Java Objects in Native Code ... 126
Exception Handling in Native Code ... 128
Behind the Scenes ... 130
Creating an Instance of a JVM in Native Code ... 130
Synchronization in Native Code ... 134

References ... 137

Index ... 139

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 3. Network Programming

What is Network Programming?

A network is a group of two or more computers (or other types of electronic devices, e.g., printers), linked together with a goal to share information. Each device linked to a network is called a *node*. A computer that is linked to a network is called a *host*. Network programming in Java involves writing Java programs that facilitate the exchange of information between processes running on different computers on a network.

Java makes it easy to write network programs. Sending a message to a process running on another computer is as simple as writing data to a local file system. Similarly, receiving a message that was sent from a process running in another computer is as simple as reading data from a local file system. Most of the programs in this chapter will involve reading and writing data over the network, and they are similar to file I/O. Please refer to the chapter on *Input/Output* for more details on file I/O. You will learn about a few new classes in this chapter that facilitate the communication between two machines on a network.

You do not need to have advanced level knowledge of networking technologies to understand or write Java programs in this chapter. This chapter covers high-level details of a few things that are involved in a network communication.

A network can be categorized based on different criteria. Based on the geographical area that a network is spread over, it is categorized as:

- Local Area Network (LAN): It covers a small area such as a building or a block of buildings.
- Campus Area Network (CAN): It covers a campus such as a university campus interconnecting multiple LANs within that campus.
- Metropolitan Area Network (MAN): It covers more geographical area than a LAN. Usually, it covers a city.
- Wide Area Network (WAN): It covers a larger geographical area such as a region of a country or multiple regions in different countries in the world.

When two or more networks are connected using routers (also known as gateways), it is called *internetworking*, and the resulting combined network is called an *internetwork,* in short, *internet* (note the lowercase i). The global internetwork, which encompasses all networks in the world connected together, is referred to as *Internet* (note the uppercase I).

Based on the topology (the arrangement of nodes in a network), a network may be categorized as a star, tree, ring, bus, hybrid, etc.

Based on the technology a network uses to transmit the data, it can be categorized as Ethernet, LocalTalk, Fiber Distributed Data Interface (FDDI), Token Ring, Asynchronous Transfer Mode (ATM), etc.

We will not cover any details about the different kinds of networks. Please refer to any standard textbook on network to learn more about network and network technologies in detail.

Communication between two processes on a computer is simple and it is achieved using inter-process communication as defined by an operating system. It is a very tedious task when two processes running on two different computers on an internet need to communicate. You need to consider many aspects of the communication before the two processes on the two computers on an internet may communicate. Some of the points that you need to consider are listed below:

- The two computers may be using different technologies, e.g., operating system, hardware, etc.
- They may be on two different networks, which use different network technologies.
- They may be separated by many other networks, which may be using different technologies. That is, two computers are not on two networks that are interconnected directly. You need to consider not just two networks, rather all networks that the data from one computer has to pass to reach another computer.
- They may be a few miles apart or in other sides of the globe. How do we transmit the information efficiently without worrying about the distance between the two computers?
- One computer may not understand the information sent by the other computer.
- The information sent over a network may be duplicated, delayed, or lost. How should the receiver and the sender handle these abnormal situations?

Simply put, two computers on a network communicate using messages (sequence of 0s and 1s). There must be well-defined rules to handle the above-mentioned (and many more) issues. The set of rules to handle a specific task is known as a *protocol*. Many types of tasks are involved in handling a network communication. There is a protocol defined to handle each specific task. There is a stack of protocols (also called protocol suite) that are used together to handle a network communication.

Modern networks are called packet switching networks, because they transmit data in chunks, which are called packets. Each packet is transmitted independent of any other packets. This makes it easy to transmit the packets from the same computer to the same destination using different routes. However, it may become a problem if a computer sends two packets to a remote computer and the second packet arrives before the first one. For this reason, each packet also has a packet number along with its destination address. There are rules to rearrange the out of order arrival of the packets at the destination computer. The following discussion will attempt to explain some of the mechanisms that are used to handle packets in a network communication.

Figure 3-1: The Internet Protocol Suite showing its five protocol layers

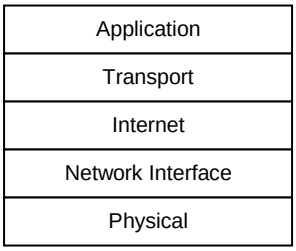


Figure 3-1 shows a layered protocol suite, which is called the Internet Reference Model or TCP/IP Layering Model. This is the most widely used protocol suite. Each layer in the model performs a well-defined task. The main advantage of having a layered protocol model is that any layer can be changed without affecting others. A new protocol can be added to any layer without changing other layers.

Each layer knows about only the layer immediately above and below it. Each layer in the protocol suite has two interfaces – one for the layer above it and one for the layer below it. For example, the transport layer has interfaces to the application layer and internet layer. That is, the transport layer

knows how to communicate only with the application layer and the internet layer. It knows nothing about the network interface layer or physical layer.

A user application (e.g. your Java program) uses the application layer to communicate to a remote application. The user application has to specify the protocol that it wants to use to communicate to a remote application. A protocol in an application layer defines the rules for formatting a message and associating the meaning to the information contained in the message, e.g., the message type, describing whether it is a request or a response, etc. After the application layer applies its formats to the message, it hands over the message to the transport layer. The examples of protocols in an application layer are Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), Gopher, Telecommunication Network (Telnet), Simple Mail Transfer Protocol (SMTP), and Network News Transfer Protocol (NNTP).

The transport layer protocol handles the ways messages are transported from one application on one computer to another application on a remote computer. It controls the data flow, error handling during data transmission, and connections between two applications. For example, a user application may hand over a very large chunk of data to the transport layer to transmit to a remote application. The remote computer may not be able to handle that large amount of data at once. It is the responsibility of the transport layer to pass a suitable amount of data at a time to the remote computer, so that the remote application can handle the data according to its capacity. The data passed to the remote computer over a network may be lost on its way due to various reasons. It is the responsibility of the transport layer to retransmit the lost data. Note that the application layer passes data to be transmitted to the transport layer only once. It is the transport layer that keeps track of the delivered or the lost data during a transmission and not the application layer. There may be multiple applications running, which are using different protocols and exchanging information with different remote applications. It is the responsibility of the transport layer to hand over messages sent to a remote application correctly. For example, you may be browsing the Internet using an HTTP protocol from one remote web server and downloading a file using an FTP protocol from another FTP server. Your computer is receiving messages from two remote computers and they are meant for two different applications running on your computer – one web browser to receive HTTP data and one FTP application to receive FTP data. It is the responsibility of the transport layer to pass incoming data to the appropriate application. You can see how different layers of the protocol suite play different roles in data transmission over the network. Depending on the transport layer protocol being used, the transport layer adds relevant information to the message and passes it to the next layer, which is the internet layer. The examples of protocols used in the transport layer are Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and Stream Control Transmission Protocol (SCTP).

The internet layer accepts the messages from the transport layer and prepares a packet suitable to be sent over the internet. It includes the Internet Protocol (IP). The packet prepared by IP is also known as an IP datagram. It consists of a header and a data area, apart from other pieces of information. The header contains the sender's IP address, destination IP address, time to live (TTL, an integer), a header checksum, and many other pieces of information specified in the protocol. The IP prepares the message into datagrams, which are ready to be transmitted over the internet. The TTL in an IP datagram header specifies how long, in terms of number of routers, an IP datagram can keep traveling before it needs to be discarded. Its size is one byte and its value could be between 1 and 255. When an IP datagram reaches a router in its route to its destination, the router decrements the TTL value by 1. If the decremented value is zero, the router discards the datagram and sends an error message back to the sender using Internet Control Message Protocol (ICMP). If the TTL value is still a positive number, the router forwards the datagram to the next router. The IP uses an address scheme, which assigns a unique address to each computer. The address is called an IP address. We will discuss IP addressing scheme in the next section in detail. The internet layer hands over the IP datagram to the next layer, which is the network interface layer. The examples of protocols in an internet layer are Internet Protocol (IP), Internet Control Message Protocol (ICMP), Internet Group Management Protocol (IGMP), and Internet Protocol Security (IPsec).

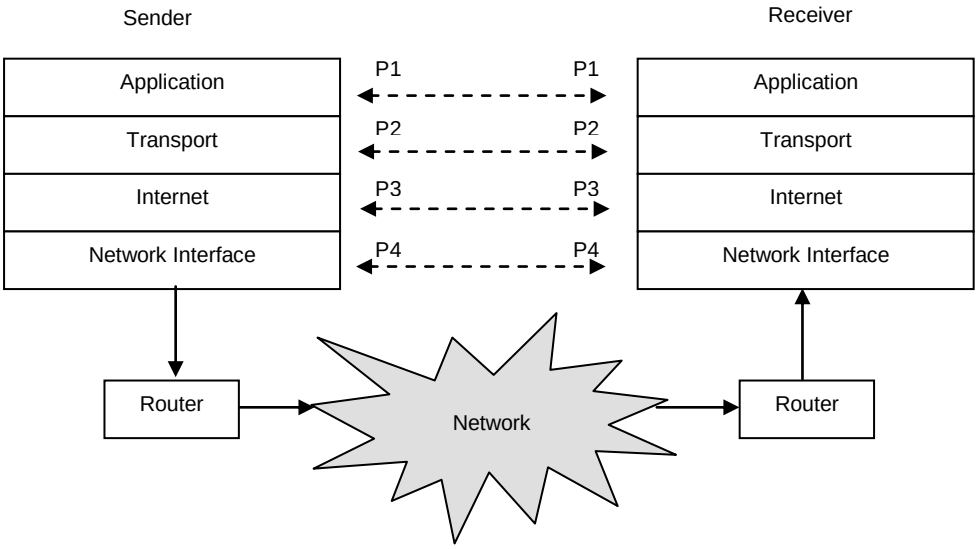
The network interface layer prepares a packet to be transmitted on the network. The packet is called a frame. The network interface layer sits just on top of the physical layer, which involves the hardware. Note that the IP layer uses the IP address to identify the destination on a network. An IP address is a virtual address, which is completely maintained by the software. The hardware is unaware of the IP address and it does not know how to transmit a frame using an IP address. The hardware must be given the hardware address, also called *Media Access Address* (MAC), of the destination that it needs to transmit the frame to. This layer resolves the destination hardware address from the IP address and places it in the frame header. It hands over the frame to the physical layer. The examples of protocols in a network interface layer are Open Shortest Path First (OSPF), Point-to-Point Protocol (PPP), Point-to-Point Tunneling Protocol (PPTP) and Layer 2 Tunneling Protocol (L2TP).

The physical layer consists of the hardware. It is responsible for converting the bits of information into signals and transmitting the signal over the wire.

TIP

Packet is a generic term that is used to mean an independent chunk of data in network programming. Each layer of protocol also uses a specific term to mean the packet it deals with. For example, a packet is called a *segment* in the TCP layer. It is called a *datagram* in the IP layer. It is called a *frame* in the network interface and physical layers. Each layer adds a header (sometimes also a trailer) to the packet it receives from the layer before it, while preparing the packet to be transmitted over the network. Each layer performs the reverse action when it receives a packet from the layer below it. It removes the header from the packet; performs some actions, if needed; and hands over the packet to the layer above it.

Figure 3-2: Transmission of packets through the protocol layers on the sender's and receiver's computers



When a packet sent by an application reaches the remote computer, it has to pass through the same layer of protocols in a reverse order. Each layer will remove its header; perform some actions, and pass the packet to the layer immediately above it. Finally, the packet reaches the remote application in the same format it started from the application on the sender's computer. Figure 3-2 shows the transmission of packets from the sender, and the receiver computer. P1, P2, P3, and P4 are the packets in different formats of the same data. A protocol layer at a destination

receives the same packet from the layer immediately below it, which the same protocol layer had passed to the layer immediately below it on the sender's computer.

IP Addressing Scheme

IP uses a unique address, called an IP address, to route an IP datagram to a destination. An IP address uniquely identifies a connection between a computer and a router. Normally, it is understood that an IP address identifies a computer. However, it should be emphasized that it identifies a connection between a computer and a router and not just a computer. A router is also assigned an IP address. A computer can be connected to multiple networks using multiple routers and each connection between the computer and the router will have a unique IP address. In such cases, a computer will be assigned multiple IP addresses and the computer is known as *multi-homed*. Multi-homing increases the availability of the network connection to a computer. In case, one network connection fails, the computer can use other available network connections.

An IP address contains two parts – a network identifier (we will call it prefix) and host identifier (we will call it suffix). The prefix identifies a network on the Internet uniquely; the suffix identifies a host uniquely within that network. It is possible to have two hosts to have IP addresses with the same suffix as long as they have a different prefix.

There are two versions of Internet Protocol – IPv4 (or simply IP) and IPv6, where v4 and v6 stand for version 4 and version 6. IPv6 is also known as Internet Protocol next generation (IPng). Note that there is no IPv5. When IP was in its full swing of popularity, it was at version 4. Before IPng was assigned a version number 6, version 5 was already assigned to another protocol called Internet Stream Protocol (ST).

Both IPv4 and IPv6 use an IP address to identify a host on a network. However, the addressing schemes in the two versions differ significantly. The next two sections will discuss addressing schemes used in IPv4 and IPv6.

Since an IP address must be unique, its assignment is controlled by an organization called *Internet Assigned Numbers Authority* (IANA). IANA assigns a unique address to each network that belongs to an organization. The organization uses the network address and a unique number to form a unique IP address for each host on the network. IANA divides the IP address allocations to five Regional Internet Registry (RIR) organizations, which allocate IP addresses in specific regions. They are as listed in Table 3-1. You can find more information on how to get a network address in your area from IANA at its website *http://www.iana.com*.

Table 3-1: List of regional Internet registries responsible for allocating network IP addresses

Regional Internet Registry Organization Name	Regions Covered
African Network Information Centre (AfriNIC)	Africa Region
Asia-Pacific Network Information Centre (APNIC)	Asia/Pacific Region
American Registry for Internet Numbers (ARIN)	North America Region
Latin American and Caribbean Internet Address Registry (LACNIC)	Latin America and some Caribbean Islands
Réseaux IP Européens Network Coordination Centre (RIPE NCC)	Europe, the Middle East, and Central Asia

IPv4 Addressing Scheme

IPv4 (or simply IP) uses a 32-bit number to represent an IP address. An IP address contains two parts – a prefix and a suffix. The prefix identifies a network and the suffix identifies a host on the network as shown in Figure 3-3. It is not easy for humans to remember a 32-bit number in binary format. IPv4 allows you to work with an alternate form using four decimal numbers. Each decimal number is in the range from 0 to 255. Programs will take care of converting four decimal numbers into a 32-bit binary number that will be used by a computer. IPv4 addresses are represented using four decimal values separated by dots, which is called dotted decimal format. Each decimal value represents the value contained in 8 bits of a 32-bit number. For example, an IPv4 address of `11000000101010000000000111100111` in a binary format can be represented as `192.168.1.231` in a dotted decimal format. The process of converting binary IPv4 to its decimal equivalent has been shown in Figure 3-4. In `192.168.1.231`, the part, `192.168.1` identifies the network address (the prefix) and the part, `231`, (the suffix) identifies the host on that network.

How do we know that `192.168.1` represents a prefix in an IPv4 address `192.168.1.231`? A rule governs the value of a prefix and a suffix in an IPv4. We will discuss how to identify a prefix and suffix in an IPv4 later in this section, when we discuss the class type of a network.

Figure 3-3: IPv4 address is a 32-bits number divided into two parts - prefix and suffix.

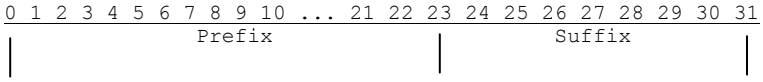


Figure 3-4: Parts of an IPv4 address in binary and decimal formats

32-bit binary representation	11000000 10101000 00000001 11100111			
Decimal Value of each octet	192	168	1	231
Parts of IPv4 address	Prefix			Suffix
Alternate Representation of IPv4	192.168.1.231			

How does an IPv4 address divide its 32 bits between a prefix and a suffix? IPv4 address space is divided in five categories called network classes. The network classes for IPv4 are named `A`, `B`, `C`, `D`, and `E`. A class type defines how many bits of the 32 bits will be used to represent the network address part of an IP address. The leading bit (or bits) in the prefix defines the class of the IP address. This is also known as a *self-identifying* or *classful* IP address, because looking at the IP address you can tell which class it belongs to.

Table 3-2: Five classes of IPv4 in the classful addressing scheme

Network Class	Prefix	Suffix	Leading Bit(s) in Prefix	Number of Networks	Number of Hosts per Network
A	8 bits	24 bits	0	128	16777214
B	16 bits	16 bits	10	16,384	65534
C	24 bits	8 bits	110	2097152	254
D	Not Defined	Not defined	1110	Not defined	Not defined
E	Not Defined	Not defined	1111	Not defined	Not defined

Table 3-2 lists the five network classes and their characteristics in IPv4. The leading bits in an IP address identify the class of the network. For example, if an IP address looks like 0XXX, where XXX is the last 31 bits of 32 bits, it belongs to the class A network; if an IP address looks like 110XXX, where XXX is the last 29 bits of 32 bits, it belongs to the class C network. There could be only 128 networks of class A type and each network can have 16777214 hosts. The number of hosts that a class A network can have is very big and it is very unlikely that a network will have that many hosts. In a class C type of network, the maximum number of hosts that a network can have is limited to 254.

What happens if an organization is assigned a network address from class C and it has only 10 hosts to attach to the network? The remaining slots in the IP addresses in that network remain unused. Recall that the host (or suffix) part in an IP address must be unique within the network (the prefix part). On the other hand, if an organization needs to connect 300 computers to a network, it needs to get two class C network addresses because getting a class B network address, which can accommodate 65534 hosts, will again waste a great many IP addresses.

Note that if the number of bits allocated for a suffix is N, the number of hosts that can be used is $2^N - 2$. Two bits patterns, all 0s and all 1s, cannot be used for a host address. They are used for special purposes. This is the reason a class C network can have a maximum of 254 hosts and not 256. Class D addresses are used as multicast addresses. Class E addresses are reserved.

The fast growth of the Internet and large number of IP addresses not being used prompted for a new addressing scheme. This scheme is simply based on one criterion that one should be able to use an arbitrary boundary between the prefix and suffix parts of an IP address, instead of predefined boundaries at 8, 16, and 24 bits. This will keep the unused addresses at a minimum. For example, if an organization needs a network number for a network with only 20 hosts, that organization can use only the 27-bit prefix and 5-bit suffix.

Two terminologies, *subnetting* and *supernetting*, are used to describe the situations when some bits from a suffix are used for the prefix, and some bits from a prefix are used as a suffix. When bits from a suffix are used as a prefix, essentially, it creates more network addresses at the cost of host addresses. The extra network addresses are called subnets. Subnetting is achieved by using a number called a *subnet mask* or an *address mask.* A subnet mask is a 32-bit number that is used to compute the network address from an IP address. Using a subnet mask eliminates the restriction that the class of a network must predefine the network number part of the IP address. A logical AND is performed on an IP address and a subnet mask. The resulting number is the network number. In this scheme of addressing, an IP address is always specified with its subnet mask. A forward slash and subnet mask follows an IP address. For example, 140.10.11.9/255.255.0.0 denotes an IP address of 140.10.11.9 with a subnet mask 255.255.0.0. It is possible to use any subnet mask whose four decimal parts ranges between 0 and 255. In our example, 140.10.11.9 is a class B address. A class B address uses 16 bits for the prefix and 16 bits for the suffix. Let us take 6 bits off the suffix and add it to the prefix. Now, our prefix is 22 bits and the suffix is only 10 bits. By doing this, we have created additional network numbers at the cost of host numbers. To describe an IP address in this scheme of subnetting, we will need to use a subnet mask of 255.255.252.0. If you write an IP address using this subnet mask as 140.10.11.9/255.255.252.0, the network address is computed as 140.10.8.0 as shown below:

```
IP Address:  10001100 00001010 00001011 00001001
Subnet Mask: 11111111 11111111 11111100 00000000
-----------------------------------------------
Logical AND: 10001100 00001010 00001000 00000000
               (140)     (10)      (8)      (0)
```

Classless Inter-Domain Routing (CIDR) is another IPv4 addressing scheme in which an IPv4 address is specified as four dotted decimal numbers along with another decimal number separated by a forward slash as `192.168.1.231/24`, where the last number, 24, denotes the prefix-length (or number of bits used for a network number) in the 32-bit IPv4 address. Note that the CIDR addressing scheme lets you define the prefix/suffix boundary at any bits in 32-bit IPv4. By moving the bits from the prefix to the suffix, you can combine multiple networks and increase the number of hosts per network. This is called *supernetting*. You can create supernets as well as subnets using CIDR notation.

Some IP addresses in an IPv4 addressing scheme are reserved for broadcast and multicast IP addresses. We will discuss broadcasting and multicasting later in this chapter.

IPv6 and its Addressing Scheme

IPv6 is a new version of IP and is the successor for IPv4. The address space in IPv4 was running out of addresses in the fast growing Internet world. IPv6 is aimed at providing enough address space, so that every computer in the world may get a unique IP address in the decades to come. Here are some of the main features of IPv6:

- IPv6 uses a 128-bit number for an IP address instead of a 32-bit number used in IPv4.
- It has different header formats for IP packets than IPv4. IPv4 has only one header per datagram, whereas IPv6 has one base header followed by multiple variable-length extension headers per datagram.
- IPv6 supports datagrams of a bigger size than IPv4.
- In IPv4, the routers performed an IP packet fragmentation. In IPv6, the sender host is supposed to perform a packet fragmentation rather than the routers. This means that the host that uses IPv6 must know in advance the path of the maximum transmission unit (MTU) (the minimum of the maximum packet size allowed by all networks to the destination host). The IP datagram's fragmentation occurs when it has to enter a network that has a lower size transmission capacity than the network the datagram is leaving. In IPv4, the fragmentation is performed by the router, which detects a lower transmission capacity network in the route. Since IPv6 allows only the host to perform the fragmentation, the host must discover the minimum size datagram that can be routed through all possible routes from the source to the destination host.
- IPv6 supports specifying routing information for the datagrams in the headers, so that routers can use it to route the datagrams through a specific route. This feature is helpful to deliver time-critical information.
- IPv6 is extensible. Any number of extension headers can be added to an IPv6 datagram, which can be interpreted in a new way.

IPv6 uses a 128-bit IP address. It uses an easy to understand notation to represent an IP address in a textual form. The 128 bits are divided into 8 fields of 16 bits each. Each field is written in hexadecimal form and separated by a colon. The following are some examples of IPv6 addresses:

```
F6DC:0:0:4015:0:BA98:C0A8:1E7
F6DC:0:0:7678:0:0:0:A21D
F6DC:0:0:0:0:0:0:A21D
0:0:0:0:0:0:0:1
```

It is common to have many fields in an IPv6 address to have zero values, especially for all Ipv4 addresses. The IPv6 address notation lets you compress contiguous fields of zero values by using two colons (`::`). You can use two colons to suppress contiguous zero value fields only once in an address. The above IPv6 address may be rewritten using the zero compression technique:

```
F6DC::4015:0:BA98:C0A8:1E7
F6DC:0:0:7678::A21D
F6DC::A21D
::1
```

Note that we could suppress only one of the two sets of contiguous zero fields in the second address `F6DC:0:0:7678::A21D`. Rewriting it as `F6DC::7678::A21D` would be invalid as it uses two colons more than once. You can use two colons to suppress contiguous zero fields, which may occur in the beginning, middle or end of the address string. If an address contains all zeros in it, you can represent it with two consecutive colons (`::`).

You can also mix hexadecimal and decimal formats while writing IPv6 addresses. The notation is useful when you have IPv4 addresses. You can write the first six 16-bit fields using a hexadecimal notation as described above and use dotted decimal notation for IPv4 for the last two 16-bit fields. The mixed notation takes a form `X:X:X:X:X:X:D.D.D.D`, where an `X` is a hexadecimal number and a `D` is a decimal number. We can rewrite the above IPv6 addresses using this notation as:

```
F6DC::4015:0:BA98:192.168.1.231
F6DC:0:0:7678::0.0.162.29
F6DC::0.0.162.29
::0.0.0.1
```

Unlike IPv4, IPv6 does not assign IP addresses based on network classes. Like IPv4, it uses CIDR addresses, so that the boundary between the prefix and suffix in an IP address can be specified at any arbitrary bit. For example, `::1` can be represented in CIDR notation as `::1/128`, where 128 represent the prefix length.

TIP
IPv6 address should be enclosed in brackets (`[]`) when it is used inside a literal string as part of a URL. This rule does not apply to IPv4. For example, if you are accessing a web server on a loopback address using Ipv4 address, you can use a URL like http://127.0.0.1/index.html. In an IPv6 address notation, you will need to use a URL like http://[::1]/index.html. Make sure your browser supports IPv6 address notation in its URL.

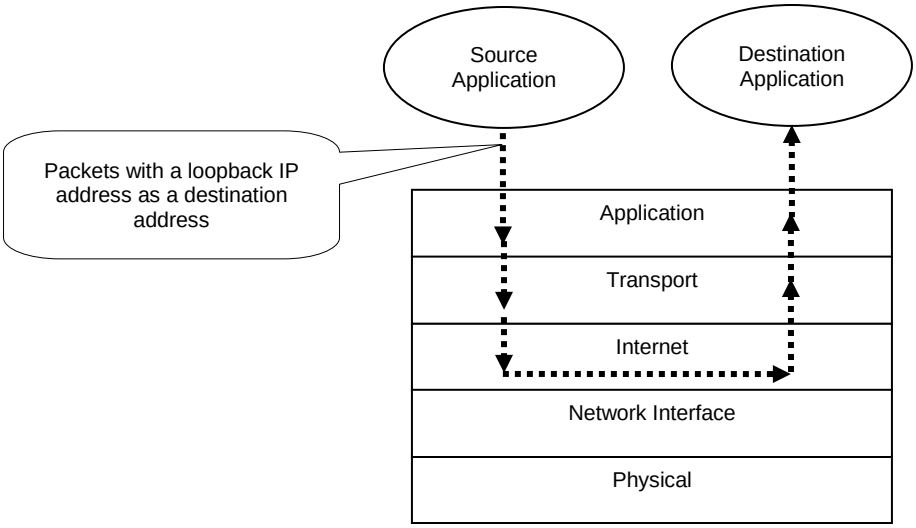
Special IP Addresses

Loopback IP Address

You need at least two computers connected using a network to test or run a network program. Sometimes, it may not be feasible or desirable to setup a network when you want to test your network program during the development phase of your project. The designer of IP realized this need. There is a provision in the IP addressing scheme to treat an IP address as a loopback address to facilitate testing of network programs using only one computer. When the Internet layer in the protocol suite detects a loopback IP address as the destination for an IP datagram, it does not pass over the packet to the protocol layer below it (that is Network Interface Layer). Rather, it turns around (or loops back and hence the name loopback address) and routes the packet back to the transport layer on the same computer. The transport layer will deliver the packet to the destination process on the same host as it would have done had the packet come from a remote host. A loopback IP address makes testing of a network program using one computer possible.

Figure 3-5 depicts the way an Internet packet, which is addressed to a loopback IP address, is processed by the IP. The packet never leaves the source computer. It is intercepted by the Internet layer and routed back to the same computer it started from.

Figure 3-5: Internet packet that has a loopback IP address as its destination is routed back to the same computer from the Internet Protocol in the Internet layer.



Loopback IP address is a reserved address and IP is required not to forward a packet with a loopback IP address as its destination address to the Network Interface layer.

In an IPv4 addressing scheme, `127.X.X.X` block is reserved for loopback addresses, where `X` is a decimal number between 0 and 255. Typically, `127.0.0.1` is used as a loopback address in IPv4. However, you are not limited to using only `127.0.0.1` as the only loopback address. If you wish, you can use `127.0.0.2` or `127.3.5.11` as a valid loopback address too. Typically, the name `localhost` is mapped to a loopback address of `127.0.0.1` on a computer.

In an IPv6 addressing scheme, there is only one loopback address, which is sufficient to perform any local testing for a network program. It is `0:0:0:0:0:0:0:1` or simply `::1`.

Unicast IP Address

Unicast is one-to-one communication between two computers on a network in which an IP packet is delivered to a single remote host. A unicast IP address identifies a unique host on a network. IPv4 as well as IPv6 support unicast IP addresses.

Multicast IP Address

Multicast is a one-to-many communication where one computer sends an IP packet that is delivered to multiple remote computers. Multicasting lets you implement the concept of group interaction such as audio or video conferencing, where one computer sends information to all computers in the group. The benefit of using multicasting in place of multiple unicasts is that the sender sends only one copy of the packet. One copy of the packet travels along the network as long it can. If receivers of the packet are on multiple networks, a copy of the packet is made when

needed, and each copy of the packet is routed independently. Finally, each receiver is delivered an individual copy of the packet. Multicasting is a very efficient way of communication between group members as it reduces network traffic.

An IP packet has only one destination IP address. How is an IP packet delivered to multiple hosts using multicasting? IP contains some addresses in its address space as multicast addresses. If a packet is addressed to a multicast address, the packet will be delivered to multiple hosts. The concept of multicast packet delivery is the same as a group membership for an activity. When a group is formed, the group is given a group ID. Any information addressed to that group ID is delivered to all group members. In a multicast communication, a multicast IP address (similar to a group ID) is used. Multicast packets are addressed to that multicast address. Each interested host registers its IP address with the local router that it is interested in communication made on that multicast address. The registration process between a host and the local router is accomplished using an Internet Group Management Protocol (IGMP). When the router receives a packet with a multicast address, it delivers a copy of the packet to each host registered with it for that multicast address. A receiver may choose to leave the multicast group any time by informing the router.

A multicast packet may travel through many routers before it finds its way to the receiver hosts. All receivers of a multicast packet may not be on the same network. There are many protocols, e.g., Distance Vector Multicast Routing Protocol (DVMRP) that deal with routing of multicast packets.

Both IPv4 and IPv6 support multicast addressing. In IPv4, Class `D` network addresses are used for multicasting. That is, the four highest order bits are `1110` in a multicast address in IPv4. In IPv6, a multicast address has the first 8 bits set to 1. That is, a multicast address in IPv6 always starts with `FF`. For example, `FF0X:0:0:0:0:0:2:0000` is an example of a multicast address in IPv6.

Anycast IP Address

Anycast is a one-to-one_from_a_group communication where one computer sends a packet to a group of computers, but the packet is delivered to exactly one computer in the group. IPv4 does not support anycasting. IPv6 supports anycasting. In anycasting, the same address is assigned to multiple computers. When a router receives a packet, which is addressed to an anycast address, it delivers the packet to the nearest computer. Anycasting is useful when a service has been replicated at many hosts and you want to provide the service at the nearest host to the client. Sometimes, anycast addressing is also called *cluster addressing*. An anycast address is used from the unicast address space. You cannot distinguish a unicast address from an anycast address by looking at their bit arrangements. When the same unicast address is assigned to multiple hosts, it is treated as an anycast address. Note that the router must know about the hosts that are assigned an anycast address, so that it can deliver the packets addressed to that anycast address to one of the nearest hosts.

Broadcast IP Address

Broadcast is a one-to-all communication where one computer sends a packet and that packet is delivered to all computers on the network. IPv4 assigns some addresses as broadcast addresses. When all 32 bits are set to 1, it forms a broadcast address and the packet is delivered to all hosts on the local subnet. When all bits in the host address are set to 1 and a network address is specified, it forms a broadcast address for the specified network number. For example, 255.255.255.255 is a broadcast address for a local subnet and `192.168.1.255` is a broadcast address for a network `192.168.1.0`. IPv6 does not have a broadcast address. You need to use a multicast address as the broadcast address in IPv6.

Unspecified IP Address

`0.0.0.0` in IPv4 and `::` in IPv6 (note that `::` denotes 128-bit IPv6 address with all bits set to zero) are known as unspecified addresses. A host uses this address as a source address to indicate that it does not have an IP address yet, e.g., during boot up process when it is not assigned an IP address yet.

Port Numbers

A port number is a 16-bit unsigned integer ranging from 0 to 65535. Sometimes, a port number is also referred to simply as a *port*. A computer runs many processes, which communicate with other processes running on remote computers. When the transport layer receives an incoming packet from the Internet layer, it needs to know which process (running in the application layer) on that computer should this packet be delivered to. A port number is a logical number that is used by the transport layer to recognize a destination process for a packet on a computer.

Each incoming packet to the transport layer has a protocol. The TCP protocol handler in the transport layer handles a TCP packet. The UDP protocol handler in the transport layer handles a UDP packet.

In the application layer, a process uses a separate protocol of each communication channel it wants to communicate on with a remote process. A process uses a unique port number for each communication channel it opens for a specific protocol and registers that port number with the specific protocol module in the transport layer. Therefore, a port number must be unique for a specific protocol. For example, a process P1 can use a port number 1988 for a TCP protocol and another process P2 can use the same port number 1988 on the same computer for a UDP protocol. A process on a host uses the protocol and the port number of the remote process to send data to the remote process.

How does a process on a computer start communicating with a remote process? For example, when you visit Yahoo's web site, you simply enter *http://www.yahoo.com* as a web page address. In this web page address, `http` indicates the application layer protocol, which uses TCP as a transport layer protocol and `www.yahoo.com` is the machine name, which is resolved to an IP address using a Domain Name System (DNS). The machine identified by www.yahoo.com may be running many processes, which may use an `http` protocol. Which process on `www.yahoo.com` does our web browser connect to? Since many people use Yahoo's web site, it needs to run its `http` service at a well-known port, so that everyone can use that port to connect to it. Typically, an `http` web server runs at port 80. You can use http://www.yahoo.com:80, which is the same as using http://www.yahoo.com. It is not always necessary to run your `http` web server at port 80. If you do not run your `http` web server at port 80, people who want to use your `http` service must know the port you are using. Internet Assigned Numbers Authority (IANA) is responsible for recommending which port numbers to use for well-known services. IANA divides the port numbers into three ranges:

- Well-known Ports: 0 -1023
- Registered Ports: 1024 - 49151
- Dynamic and/or Private Ports: 49152 - 65535

Well-known port numbers are used by most commonly used services provided globally such as HTTP, FTP, etc. Table 3-3 lists some of the well-known ports that are used for well-known

application layer protocols. Generally, you need administrative privileges to use a well-known port on a computer.

Table 3-3: A partial list of well-known ports used for some application layer protocols

Application Layer Protocol	Port Number
echo	7
FTP	21
Telnet	23
SMTP	25
HTTP	80
POP3	110
NNTP	119

An organization (or a user) can register a port number with IANA in the registered ports range to be used by an application. For example, 1099 (TCP/UDP) port has been registered for the RMI Registry (RMI stands for Remote Method Invocation).

Any application can use a port number from dynamic/private port number range.

Socket API and Client-Server Paradigm

We have not yet started discussing Java classes that make network communication possible in a Java program. In this section, we will cover sockets and the client-server paradigm that is used in a network communication between two remote hosts.

We covered briefly the different lower layers of protocols and their responsibilities in the previous sections. It is time to move up in the protocol stack and discuss the interaction between the application layer and the transport layer. How does an application use these protocols to communicate with a remote application? Operating systems provide an Application Program Interface (API) called a socket, which lets two remote applications communicate taking advantage of lower level protocols in the protocol stack. A socket is not another layer of protocol. It is an interface between the transport layer and the application layer. It provides a standard way of communication between the two layers, which in turn provides a standard way of communication between two remote applications.

There are two kinds of sockets – a *connection-oriented socket* and a *connectionless socket*. A connection-oriented socket is also called a *stream socket*. A connectionless socket is also called a *datagram socket*. Note that the data is always sent from one host to another on the Internet using IP datagrams one datagram at a time.

Transmission Control Protocol (TCP), which is used in a transport layer, is one of the most widely used protocols to provide connection-oriented sockets. The application hands over data to a TCP socket and the TCP takes care of streaming the data to the destination host. The TCP takes care of all issues like ordering, fragmentation, assembly, lost data detection, duplicates data transmission, etc., on both sides of the communication, which gives the impression to the applications that data is flowing like a continuous stream of bytes from the source application to the destination application. No physical connection at the hardware level exists between two hosts that

use TCP sockets. It is all implemented in software. Sometimes, it is also called a *virtual connection*. The combination of two sockets uniquely defines a connection.

In a connection-oriented socket communication, a client and a server create a socket at their ends; establish a connection, and exchange information. TCP takes care of the errors that may occur during data transmission. TCP is also known as a reliable transport level protocol, because it guarantees the delivery of the data. If it could not deliver data for some reasons, it will inform the sender application about the error conditions. After it sends the data, it waits for an acknowledgment from the receiver to make sure that the data reached its destination. However, the reliability that TCP offers comes at a price. The overhead as compared to a connectionless protocol is much more significant, and it is slower. TCP makes sure that a sender sends the amount of data to the receiver, which can be handled by the receiver's buffer size. It also handles traffic congestion over the network. It slows down the data transmission when it detects traffic congestion. Java supports TCP sockets.

User Datagram Protocol (UDP), which is used in a transport layer, is the most widely used protocol that provides a connectionless socket. It is unreliable, but much faster. It lets you send limited sized data one packet at a time, unlike TCP that lets you send data as a stream of any size and it handles the details of segmenting them in appropriate size of packets. In short, we can say that nothing is guaranteed when you send data using UDP. However, it is still used in many applications and it works very well. The sender sends a UDP packet to a destination and forgets about it. If receiver gets it, it gets it. Otherwise, there is no way to know - for the receiver - that there was a UDP packet sent to it. You can compare the communication used in TCP and UDP to the communication used in a telephone and mailing a letter. A telephone conversation is reliable and it offers acknowledgment between two parties that are communicating. When you mail a letter, you do not know when the addressee receives it, or if he received it at all. There is another important difference between UDP and TCP. UDP does not guarantee the ordering of data. That is, if you send five packets to a destination using UDP, those five packets may arrive in any order. However, TCP guarantees that packets will be delivered in the order they were sent. Java supports UDP sockets.

Which protocol should you use: TCP or UDP? It depends on how the application will be used. If data integrity is of utmost significance, you should use TCP. If speed is prioritized over lower data integrity, you should use UDP. For example, a file transfer application should use TCP, whereas a video conferencing application should use UDP. If you lose video data of a few pixels, it does not matter much to the video conference. It can continue. However, if you lose a few bytes of data when a file is being transferred, that file may not be usable at all.

How do two remote applications start communicating? Which application initiates the communication? How does an application know that a remote application is interested in communicating with it? Have you ever dialed a customer service number of a company to talk to a customer service representative? If you have talked to a company's customer service representative, you already have experienced two remote applications communicate. We will refer to the mechanism of using a company's customer service to explain remote communication in this section. You and a company's representative are at two remote locations. You need a service and the company provides that service. In other words, you are the client and the company is a service provider (or a server). You do not know when you will need a service from the company. The company provides a customer service phone number, so that you can contact the company. There is one more thing the company does. What is it that the company must do to provide you a service? Can you guess? It waits for your calls at the phone number that it gave you. The communication has to happen between you and the company, and the company has already taken one step forward in that communication by *passively* waiting for your call. As soon as you dial the company's number, a connection is established and you exchange information with the company's representative. Both of you hang up, at the end, to discontinue the communication. The network communication using sockets is similar to the communication that happens between you and a

company's representative. If you understand this example of communication, understanding sockets is easy.

Two remote applications use a pair of sockets to communicate. You need two endpoints for any communication to occur. A socket is a communication endpoint on each side of the communication channel. Communication over a pair of sockets follows a typical client-server communication paradigm. One application creates a socket and passively waits to be contacted by another remote application. The application, which waits for a remote application to contact it, is called a *server application* or simply a *server*. Another application creates a socket and initiates the communication with the waiting server application. This is called a *client application* or simply a *client*. Many other steps must be performed, before a client and a server can exchange information. For example, a server must advertise the location and other details about itself, so that a client may contact it.

A socket passes through different states. Each state marks an event. It is the state of the socket that tells us what a socket can do and what it cannot do. Generally, a socket's lifecycle is described by eight primitives listed in Table 3-4. The following text elaborates each socket primitive in detail.

Table 3-4: List of typical socket primitives and their descriptions

Primitives	Description
`Socket`	Creates a socket, which is used by an application to serve as a communication endpoint
`Bind`	Associates a local address to the socket. The local address includes an IP address and a port number. The port number must be a number between 0 and 65535. It should be unique for the protocol being used for the socket on the computer. For example, if a TCP socket uses port 12456, a UDP socket can also use the same port number 12456.
`Listen`	It defines the size of its wait-queue for a client request. It is performed only by a connection-oriented server socket.
`Accept`	Waits for a client request to arrive. It is performed only by a connection-oriented server socket.
`Connect`	Attempts to establish a connection to a server socket, which is waiting on `accept` primitive. It is performed by a connection-oriented client socket.
`Send/Sendto`	Sends data. Usually `send` indicates a send operation on a connection-oriented socket and `Sendto` indicates a send operation on a connectionless socket.
`Receive/ReceiveFrom`	Receives data. They are counterparts of `Send` and `Sendto`.
`Close`	Closes a connection

- `Socket`: A server creates a socket by specifying what kind of socket it is – a stream socket or a datagram socket.
- `Bind`: It associates the socket to a local IP address and a port number. Note that a host can have multiple IP addresses. A socket can be bound to one of the IP addresses of the host or all of them. Binding a socket to all available IP addresses for the host is also known as binding to a wild-card address. Binding reserves the port number for this socket. No other socket can use that port number for communication. The bound port will be used by the transport protocol (TCP as well as UDP) to route the data intended for this socket. We will explain more about transferring data between the transport layer and a socket little later in this section. For now, it is enough to understand that in binding, the socket tells the transport layer that here is my IP

address and port number, and if you get any data addressed to this address, please pass that data to me. The IP address and the port number to which a socket is bound are called a *local address* and a *local port* for the socket, respectively.

- `Listen`: A server informs the operating system to place the socket in a passive mode, so that it waits for the incoming client requests. A server also specifies a wait queue size for the socket. When a client contacts the server at this socket, the client request is placed in that queue. Initially, the queue is empty. If a client contacts the server at this socket and the wait queue is full, the client's request is rejected. At this point, the server is not yet ready to accept any client request.
- `Accept`: A server informs the operating system that this socket is ready to accept client requests. This step is not performed if the server is using a socket using a connectionless transport protocol such as UDP. This step is performed for TCP server sockets. When a socket sends an accept message to the operating system, it blocks until it receives a client request for a new connection.
- `Connect`: This is the most important phase in a connection-oriented socket communication. Only a connection-oriented client socket performs this step. The client socket sends a request to the server socket to establish a connection. The server socket has issued `accept` and has been waiting for a client request to arrive. The client socket sends the IP address and the port number of the server socket. Recall that a server socket binds an IP address and a port number before it starts listening and accepting connections from outside. Along with its request, a client socket also sends its own IP address and the port number to which it is already bound.

  An important question arises at this point. How does the transport layer (e.g. TCP) know that the packet (in the form of a request for a connection) that came from a client has to be handed over to the server socket? During the binding phase, a socket specifies its local IP address and a local port number as well as a remote IP address and a remote port number. If a server socket wants to accept a connection only from a specific remote host IP address and port number, it can do so. Usually, a server socket will accept a connection from any client and it will specify an unspecified IP address and a zero port number as its remote address. A server socket passes these five pieces of information - a local IP address, a local port number, a remote IP address, and a remote port number, and a buffer, to the transport layer. The Transport layer stores them for future use in a special structure, called a Transmission Control Block (TCB). When a packet from outside arrives at the transport layer, it looks up its TCB based on the four pieces of information contained in the incoming packet - <<source IP address, source port number, destination IP address, destination port number>>. Recall that the client sends the source and destination addresses in each TCP packet to the server. The transport layer attempts to find a buffer that is associated with the source and destination addresses. If it finds a buffer, it transfers the incoming data to the buffer and notifies the socket that there is some information for it in the buffer. If a server socket is accepting requests from any client (all zeros in the remote address), the data from any client will be routed to its buffer.

  Once a server socket detects a request from a client, it creates a new socket with the remote client's address information. The new socket is bound using a <<local IP address, local port number (the same as server socket's port number), remote IP address, and remote port number>> and a new buffer is created and bound to this combined addresses. In fact, two buffers are created for a socket – one for the incoming data and one for the outgoing data. At this point, a server socket lets the new socket communicate with the client socket that requested a connection. The server socket itself can close itself (accepting no more client requests for a connection) or it can start waiting again to accept another client request for a connection.

  After a connection is established, between the two sockets – a client and a server, they can exchange information. A TCP connection supports full duplex connection. That is, data can be sent or received in both directions simultaneously.

A client socket knows its local IP address, local port number, remote IP address, and remote port number before it attempts to connect to a server. At the client end, the creation of a TCB follows similar rules.

Once the client and server sockets are in place, two sockets, the client socket and the server socket dedicated to the client, define a connection.

- `Send/Sendto`: It is the stage when a socket sends data.
- `Receive/ReceiveFrom`: It is the stage when a socket receives data.
- `Close`: It is time to say goodbye. Finally, the server and client sockets close the connection.

TIP
A server socket acts like a receptionist sitting at the front desk in an office (server). A client comes in and talks to the receptionist first. A connection request comes from a client to the server and contacts the server socket first. The receptionist hands over the client to another staff. At this point, the job of the receptionist is over with that client. She continues her work of waiting to welcome another client coming to the office. Meanwhile, the first client can continue talking to another staff as long as he needs. Similarly, the server socket creates a new socket and assigns that new socket to the client for any further communication. As soon as a server socket assigns a new socket to the client, its job is over with that client. It will wait for another incoming request for connection from another client. Note that apart from many other details, a socket has five important pieces of information associated with it – a protocol, a local IP address, a local port number, a remote IP address and a remote port number.

Subsequent sections will discuss Java classes that support different kinds of sockets to facilitate network programming. Java classes that are related to network programming are in java.net, javax.net and `javax.net.ssl` packages.

Representing a Machine Address

Internet protocol uses the IP addresses of machines to deliver packets. Using IP addresses in a program is not always easy, because of its numeric format. You may be able to memorize and use IPv4 addresses, because it is only four decimal numbers in length. Memorizing and using IPv6 addresses is a little more difficult, because they are eight numbers in a hexadecimal format. Every computer also has a name such as www.yahoo.com. Using a computer name in your program makes your life much easier. Java provides classes that let you use a computer name or an IP address in a Java program. If you use a computer name, Java takes care of resolving the computer name to its IP address using a Domain Name System (DNS).

An object of the `InetAddress` class represents an IP address. It has two subclasses – `Inet4Address` and `Inet6Address`, which represent IPv4 and IPv6 addresses respectively. The `InetAddress` class does not have a public constructor. It provides four factory methods to create its object. They are as follows. All of them throw a checked `UnknownHostException`.

```
public static InetAddress[] getAllByName(String host)
public static InetAddress getByAddress(byte[] addr)
public static InetAddress getByAddress(String host, byte[] addr)
public static InetAddress getByName(String host)
```

The `host` argument refers to a computer name or IP address in the standard format. The `addr` argument refers to the parts of an IP address as a byte array. If you specify an IPv4 address, `addr`

must be a four-element `byte` array. For IPv6 addresses, it should be an eight-element `byte` array. The `InetAddress` class takes care of resolving the host name to an IP address using DNS.

Sometimes, a host may have multiple IP addresses. The `getAllByName()` method returns all addresses as an array of `InetAddress` objects.

Typically, you create an object of the `InetAddress` class using one of the above four factory methods and pass that object to other methods during a socket creation and connection. The following snippet of code demonstrates some of its uses. You will need to handle exceptions when you use the `InetAddress` class or its subclasses.

```
// Get the IP address of yahoo web server
InetAddress yahooAddress = InetAddress.getByName("www.yahoo.com");

// Get the loopback IP address
InetAddress loopbackAddress = InetAddress.getByName(null);

// Get the address of the local host. Typically, a name "localhost" is
// mapped to a loopback address. Here, we are trying to get the IP
// address of the local computer where this code executes and
// not the loopback address
InetAddress myComputerAddress = InetAddress.getLocalHost();
```

Listing 3-1 demonstrates the use of the `InetAddress` class and some of its methods.

Listing 3-1: Demonstrating the use of the InetAddress class

```
// InetAddressTest.java
package com.jdojo.chapter3;

import java.net.InetAddress;

public class InetAddressTest {
     public static void main(String[] args) {
          // Print www.yahoo.com address details
          printAddressDetails("www.yahoo.com");

          // Print the loopback address details
          printAddressDetails(null);

          // Print the loopback address details using IPv6 format
          printAddressDetails("::1");
     }

     public static void printAddressDetails(String host) {
          int timeOutinMillis = 10000;
          System.out.println("Host: " + host + " details starts...");
          try {
               InetAddress addr = InetAddress.getByName(host);
               System.out.println("Host IP Address: " +
                                  addr.getHostAddress());
               System.out.println("Canonical Host Name: " +
                                  addr.getCanonicalHostName());
               System.out.println("isReachable(): " +
                                  addr.isReachable(timeOutinMillis));
               System.out.println("isLoopbackAddress(): " +
```

```
                                   addr.isLoopbackAddress());
            }
            catch (Exception e) {
                 e.printStackTrace();
            }
            finally {
                 System.out.println("Host: " + host + " details ends...\n");
            }
       }
}
```

Output: (You may get a different output.)

```
Host: www.yahoo.com details starts...
Host IP Address: 209.191.93.52
Canonical Host Name: f1.www.vip.mud.yahoo.com
isReachable(): false
isLoopbackAddress(): false
Host: www.yahoo.com details ends...

Host: null details starts...
Host IP Address: 127.0.0.1
Canonical Host Name: localhost
isReachable(): true
isLoopbackAddress(): true
Host: null details ends...

Host: ::1 details starts...
Host IP Address: 0:0:0:0:0:0:0:1
Canonical Host Name: HYE6754
isReachable(): true
isLoopbackAddress(): true
Host: ::1 details ends...
```

Representing a Socket Address

A socket address contains two parts – an IP address and a port number. An object of the `InetSocketAddress` class represents a socket address. You can use any of the following three constructors to create an object of the `InetSocketAddress` class.

```
public InetSocketAddress(InetAddress addr, int port)
public InetSocketAddress(int port)
public InetSocketAddress(String hostname, int port)
```

All constructors will attempt to resolve a host name to an IP address. If a host name could not be resolved, the socket address will be flagged as unresolved, which you can test using its `isUnresolved()` method. If you do not want this class to resolve the address when creating its object, you can use the following factory method to do so.

```
public static InetSocketAddress createUnresolved(String host, int port)
```

Listing 3-2 shows how to create resolved and unresolved `InetSocketAddress` objects. The `getAddress()` method returns an `InetAddress` object. If a host name is not resolved, the `getAddress()` method returns `null`. If you use an unresolved `InetSocketAddress` object with a socket, an attempt is made to resolve the host name during the bind process.

Listing 3-2: Creating an InetSocketAddress object

```
// InetSocketAddressTest.java
package com.jdojo.chapter3;

import java.net.InetSocketAddress;

public class InetSocketAddressTest {
     public static void main(String[] args) {
          InetSocketAddress addr1 = new InetSocketAddress("::1", 12889);
          printSocketAddress(addr1);

          InetSocketAddress addr2 =
                    InetSocketAddress.createUnresolved("::1", 12881);
          printSocketAddress(addr2);
     }

     public static void printSocketAddress(InetSocketAddress sAddr) {
          System.out.println("Socket Address: " + sAddr.getAddress());
          System.out.println("Socket Host Name: " + sAddr.getHostName());
          System.out.println("Socket Port: " + sAddr.getPort());
          System.out.println("isUnresolved(): " + sAddr.isUnresolved());
          System.out.println();
     }
}
```

```
Output:

Socket Address: /0:0:0:0:0:0:0:1
Socket Host Name: HYE6754
Socket Port: 12889
isUnresolved(): false

Socket Address: null
Socket Host Name: ::1
Socket Port: 12881
isUnresolved(): true
```

Creating a TCP Server Socket

An object of the `ServerSocket` class represents a TCP server socket in Java. A `ServerSocket` object is used to accept a connection request from a remote client. The `ServerSocket` class provides many constructors. You can use the no-args constructor to create an unbound server socket and use its `bind()` method to bind it to a local port and a local IP address.

```
// Create an unbound server socket
ServerSocket serverSocket = new ServerSocket();
```

```
// Create a socket address object
InetSocketAddress endPoint = new InetSocketAddress("localhost", 12900);

// Set the wait queue size to 100
int waitQueueSize = 100;

// Bind the server socket to localhost and at port 12900 with a wait
// queue size of 100
serverSocket.bind(endPoint, waitQueueSize);
```

There is no separate `listen()` method in the `ServerSocket` class that corresponds to the `listen` socket primitive. Its `bind()` method takes care of specifying the waiting queue size for the socket.

You can combine `create`, `bind`, and `listen` operations in one step by using any of the following constructors of the `ServerSocket` class. The default value for the wait queue size is 50. The default value for a local IP address is the wild-card address, which means all IP addresses of the server machine.

```
public ServerSocket(int port)
public ServerSocket(int port, int waitQueueSize)
public ServerSocket(int port, int waitQueueSize, InetAddress bindAddr)
```

We can combine socket creation and bind steps described in the above snippet of code into one statement as shown below.

```
// Create a server socket at port 12900, with 100 as the wait queue size
// and at the localhost loopback address
ServerSocket serverSocket =
     new ServerSocket(12900, 100, InetAddress.getByName("localhost"));
```

Once a server socket is created and bound, it is ready to accept incoming connection requests from remote clients. To accept a remote connection request, you need to call the `accept()` method on the server socket. The `accept()` method call blocks until a request from a remote client arrives in its wait queue. When the server socket receives a request for a connection, it reads the remote IP address and the remote port number from the request and creates a new *active* socket. The newly created active socket is returned as the return value from the `accept()` method. An object of the `Socket` class represents the new active socket. We state that the `accept()` method returns a new *active* socket, because it is not a *passive* socket like a server socket, which waits for a remote request. It is an active socket, because it is created for an active communication with the remote client. Sometimes, this active socket is also called a *connection socket,* because it handles the data transmission on a connection.

```
// Wait for a new remote connection request
Socket activeSocket = serverSocket.accept();
```

Once the server socket returns from the `accept()` method call, the number of sockets in the server application increases by one. You have one passive server socket and one more active socket. The new active socket is the end-point at the server for the new client connection. At this point you need to handle the communication with the client using the new active socket.

At this point, you are ready to read and write data on the connection represented by the new socket. A Java TCP socket provides a full duplex connection. It lets you read data from the connection as well as write data to the connection. The `Socket` class provides two methods -

`getInputStream()` and `getOutputStream()`. The `getInputStream()` method returns an `InputStream` object that you can use to read data from the connection. The `getOutputStream()` method returns an `OutputStream` object, which you can use to write data to the connection. You would use `InputStream` and `OutputStream` objects as if you are reading from and writing to a file on a local file system. It is assumed that you are familiar with Java I/O. If you are not familiar with Java I/O, please refer to the chapter on *Input/Output* (Vol. 2), before you proceed in this section. However, you can still read about the UDP socket in the section later in this chapter. When you are done with reading/writing data on the connection, you would close the `InputStream/OutputStream`, and finally close the socket. The following snippet of code reads a message from a client, and echoes it to the client. Note that the server and the client must agree on the format of the message, before they start communicating. We assume that the client sends one line of text at a time.

```
// Create a buffered reader and a buffered writer from the socket's input
// and output streams, so that we can read/write one line at a time
BufferedReader br = new BufferedReader( new InputStreamReader(
                                          activeSocket.getInputStream()
                                        )
                                      );

BufferedWriter bw = new BufferedWriter( new OutputStreamWriter(
                                          activeSocket.getOutputStream()
                                        )
                                      );
```

You can use `br` and `bw` exactly the same way you would use them to read from a file or write to a file. An attempt to read from an input stream blocks until data becomes available on the connection.

```
// Read one line of text from the connection
String inMsg = br.readLine();

// Write some text to the output buffer
bw.write('hello from server");
bw.flush();
```

At the end, close the connection using the socket's `close()` method. Closing the socket also closes its input and output streams. In fact, you can close one of the three – input stream, output stream or the socket, and the other two will be closed automatically. Any attempts to read/write on a closed socket results in an exception. You can check if a socket is closed by using its `isClosed()` method, which returns `true` if the socket is closed.

```
// Close the socket
activeSocket.close();
```

TIP
Once you close a socket, you cannot reuse it. You must create a new socket and bind it before using it.

A server handles two kinds of work – accepting new connection requests and responding to already connected clients. If responding to a client takes a very small amount of time, you can use the strategy as shown below:

```
ServerSocket serverSocket = create a server socket here;
while(true) {
     Socket activeSocket = serverSocket.accept();
     // Handle the client request on activeSocket here...
}
```

The above strategy handles one client at a time. It is suitable only if the number of concurrent incoming connections is very low and a client's request takes a very small amount of time to respond. If a client request takes a significant amount of time to respond, all other clients will have to wait before they can be served.

Another strategy to work with multiple client requests would be to handle each client's request in a separate thread, so that the server can serve multiple clients at the same time. The following pseudo code outlines this strategy.

```
ServerSocket serverSocket = create a server socket here;
while(true) {
     Socket activeSocket = serverSocket.accept();
     Runnable runnable = new Runnable() {
                              public void run() {
                                   // Handle the client request on
                                   // the activeSocket here...
                              }
                         };
     new Thread(runnable).start(); // start a new thread
}
```

The above strategy seems to work fine, until you have too many threads that are created for concurrent client connections. Another strategy that works well in most of the situations is to have a thread pool to serve all client connections. If all threads in a pool are busy serving clients, the request should wait until a thread becomes free to serve it.

Listing 3-3 has complete code for an echo server. It creates a new thread to handle each client request.

Listing 3-3: An echo server based on TCP sockets

```
// TCPEchoServer.java
package com.jdojo.chapter3;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;

public class TCPEchoServer {
     public static void main(String[] args) {
          try {
               // Create a Server socket
               ServerSocket serverSocket = new ServerSocket(12900, 100,
                         InetAddress.getByName("localhost"));
               System.out.println("Server started at: " + serverSocket);
```

```
            // Keep accepting client connections in an infinite loop
            while (true) {
                System.out.println("Waiting for a connection ...");

                // Accept a connection
                final Socket activeSocket = serverSocket.accept();

                System.out.println("Received a connection from " +
                                   activeSocket);

                // Create a new thread to handle the new connection
                Runnable runnable = new Runnable() {
                    public void run() {
                        // Handle the client request
                        handleClientRequest(activeSocket);
                    }
                };
                new Thread(runnable).start(); // start a new thread
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void handleClientRequest(Socket socket) {
        BufferedReader socketReader = null;
        BufferedWriter socketWriter = null;

        try {
            // Create a buffered reader and writer using
            // the socket's input and output streams
            socketReader = new BufferedReader(new InputStreamReader(
                                socket.getInputStream()));

            socketWriter = new BufferedWriter(new OutputStreamWriter(
                                socket.getOutputStream()));

            String inMsg = null;
            while ((inMsg = socketReader.readLine()) != null) {
                System.out.println("Received from client: " + inMsg);

                // Echo the received message to the client
                String outMsg = inMsg;
                socketWriter.write(outMsg);
                socketWriter.write("\n");
                socketWriter.flush();
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            try {
                socket.close();
            }
            catch (Exception e) {
```

```
                    e.printStackTrace();
                }
            }
        }
}
```

Creating a TCP Client Socket

An object of the `Socket` class represents a TCP client socket. We have already seen how an object of the `Socket` class works with a TCP server socket. For a server socket, we got an object of the `Socket` class as the return value from the server socket's `accept()` method. For a client socket, we will have to perform three additional steps – create, bind, and connect. The `Socket` class provides many constructors that let you specify the remote IP address and port number. These constructors bind the socket to a local host and an available port number.

```
// Create a client socket, which is bound to the local host at any
// available port; connected to remote IP of 192.168.1.2 at port 3456
Socket socket = new Socket("192.168.1.2", 3456);

// Create an unbound client socket. bind it and connect it
Socket socket = new Socket();
socket.bind(new InetSocketAddress("localhost", 14101));
socket.connect(new InetSocketAddress("localhost", 12900));
```

Once you get a connected `Socket` object, you can use its input stream and output stream using the `getInputStream()` and `getOutputStream()` methods respectively. You can read/write on the connection the same way you would read/write from/to a file.

Listing 3-4 contains the complete code for an echo client application. It receives input from the user; sends the input to the echo server as listed in Listing 3-3; and prints the server's response on the standard output. Both applications, the echo server and the echo client, must agree on the format of the messages that they will be exchanging. They exchange one line of text at a time. It is important to note that we must append a new line with every message that is sent across the connection, because we are using the `readLine()` method of the `BufferedReader` class, which returns only when it encounters a new line. The client application must use the same IP address and port number where the server socket is accepting the connection.

Listing 3-4: An echo client based on TCP sockets

```
// TCPEchoClient.java
package com.jdojo.chapter3;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.Socket;

public class TCPEchoClient {
     public static void main(String[] args) {
          Socket socket = null;
          BufferedReader socketReader = null;
```

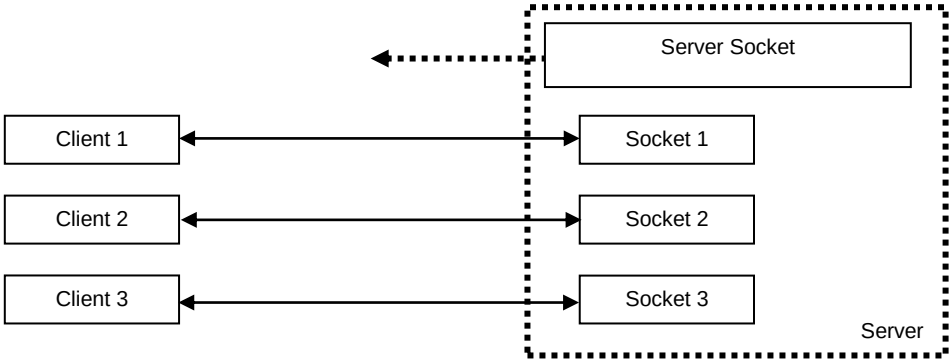
```
BufferedWriter socketWriter = null;
try {
     // Create a socket that will connect to localhost
     // at port 12900. Note that the server must also be running
     // at localhost and 12900
     socket = new Socket("localhost", 12900);

     System.out.println("Started client socket at " +
                        socket.getLocalSocketAddress());

     // Create a buffered reader and writer using
     // the socket's input and output streams
     socketReader = new BufferedReader(new InputStreamReader(
                                socket.getInputStream()));
     socketWriter = new BufferedWriter(new OutputStreamWriter(
                                socket.getOutputStream()));

     // Create a buffered reader for user's input
     BufferedReader consoleReader =
          new BufferedReader(new InputStreamReader(System.in));

     String promptMsg = "Please enter a message (Bye to quit):";
     String outMsg = null;

     System.out.print(promptMsg);
     while ((outMsg = consoleReader.readLine()) != null) {
          if (outMsg.equalsIgnoreCase("bye")) {
               break;
          }

          // Add a new line to the message to the server,
          // because the server reads one line at a time
          outMsg = outMsg;
          socketWriter.write(outMsg);
          socketWriter.write("\n");
          socketWriter.flush();

          // Read and display the message from the server
          String inMsg = socketReader.readLine();
          System.out.println("Server: " + inMsg);

          System.out.println(); // Print a blank line
          System.out.print(promptMsg);
     }
}
catch (Exception e) {
     e.printStackTrace();
}
finally {
     // Finally close the socket
     if (socket != null) {
          try {
               socket.close();
          }
          catch (IOException e) {
               e.printStackTrace();
          }
```

```
                }
            }
        }
}
```

Putting a TCP Server and TCP Clients Together

Figure 3-6 shows the setup in which three clients are connected to a server. Two `Socket` objects – one at each end, represents a connection. The `ServerSocket` object in the server keeps waiting for incoming connection requests from a client.

Figure 3-6: Client-Server setup using ServerSocket and Socket objects. ServerSocket waits for incoming connections. Two Socket objects represent an established connection – one at each end of the connection.



Listing 3-3 and Listing 3-4 list the complete program for a TCP echo server and client application. You need to run the `TCPEchoServer` class first, and then the `TCPEchoClient` class. The server application waits for a client application to connect. The client application prompts the user to enter a text message on the console. Once the user enters a text message and presses the Enter key, the client application sends that text to the server. The server responds back with the same message. Both applications print the details about the conversation on the standard output. The following are the outputs for an echo server and an echo client. You can run multiple instances of the `TCPEchoClient` application. The server application handles each client connection in a separate thread.

```
Server Console Output:

Server started at:
ServerSocket[addr=localhost/127.0.0.1,port=0,localport=12900]
Waiting for a connection ...
Received a connection from
Socket[addr=/127.0.0.1,port=1698,localport=12900]
Waiting for a connection ...
Received from client: Hello


Client Console Output:

Started client socket at /127.0.0.1:1698
Please enter a message (Bye to quit):Hello
```

```
Server: Hello

Please enter a message (Bye to quit):Bye
```

Working with UDP Sockets

A socket based on the User Datagram Protocol (UDP) is connectionless and is based on datagrams, as opposed to a TCP socket, which is connection-oriented and is based on streams. The effect of being a connectionless socket is that the two sockets (client and server) do not establish a connection before they communicate. Recall that TCP has a server socket and its sole function was to listen for a connection request from remote clients. Since UDP is a connectionless protocol, there will not be a server socket when we work with UDP. In TCP sockets, the impression of having a stream oriented data transmission between the client and server was produced by TCP in the transport layer, because of its connection-oriented features. TCP maintained the state of the data being transmitted on each side of the connection. The implication of UDP being a connectionless protocol is that each side (client and server) sends or receives a chunk of data without any prior knowledge of communication between them. In a communication using UDP, each chunk of data that is sent to the same destination is independent of the data sent prior to it. The chunk of data that is sent using UDP is called a datagram or a UDP packet. Each UDP packet has the data, the destination IP address, and the destination port number. UDP is an unreliable protocol, because it does not guarantee the delivery and the order of delivery of packets to the intended recipient.

TIP
Although UDP is a connectionless protocol, you can build a connection-oriented communication using UDP in your application. You will need to write the logic that will handle the lost packets, out of order packet delivery and many more things. Note that TCP provides you all these features at transport layer and your application does not have to worry about them.

Writing an application that uses UDP sockets is easier than writing an application that uses TCP sockets. You have to deal with only two classes while using UDP sockets – `DatagramPacket` class and `DatagramSocket` class. An object of the `DatagramPacket` class represents a UDP datagram, which is the unit of data transmission over a UDP socket. An object of the `DatagramSocket` class represents a UDP socket, which is used to send or receive a datagram packet. Here are the steps you need to perform to work with UDP sockets:

- Create an object of the `DatagramSocket` class and bind it to a local IP address and a local port number.
- Create an object of the `DatagramPacket` class, which represents a datagram packet.
- Use the `send()` method to send the datagram packet to its destination. On the receiving end, use the `receive()` method to read the datagram packet.

You can use one of the constructors to create an object of the `DatagramSocket` class. All of them will create the socket and bind it to a local IP address and a local port number. Note that a UDP socket does not have a remote IP address and a remote port number, because it is never connected to a remote socket. It can receive/send a datagram packet from/to any UDP socket.

```
// Create a UDP Socket bound to a port number 15900 at localhost
DatagramSocket udpSocket = new DatagramSocket(15900, ”localhost”);
```

The `DatagramSocket` class provides a `bind()` method, which lets you bind the socket to a local IP address and a local port number. Typically, you do not need to use this method as you specify the socket address to which it needs to be bound in its constructor, as we did in the above code.

A `DatagramPacket` contains three things – a destination IP address, a destination port number, and the data. The constructors for the `DatagramPacket` class fall into two categories. Constructors in one category let you create a `DatagramPacket` object to receive a packet. They require only the buffer size, offset and length of data in that buffer. Constructors in the other category let you create a `DatagramPacket` object to send a packet. They require you to specify the destination address along with the data. If you have created a `DatagramPacket` without specifying the destination address, you can set the destination address afterwards using its `setAddress()` and `setPort()` methods.

Constructors of the `DatagramPacket` class to create a packet to receive data are as follows:

```
public DatagramPacket(byte[] buf, int length)
public DatagramPacket(byte[] buf, int offset, int length)
```

Constructors of the `DatagramPacket` class to create a packet to send data are as follows:

```
public DatagramPacket(byte[] buf, int length,
                      InetAddress address, int port)
public DatagramPacket(byte[] buf, int offset,
                      int length, InetAddress address, int port)
public DatagramPacket(byte[] buf, int length, SocketAddress address)
public DatagramPacket(byte[] buf, int offset,
                      int length, SocketAddress address)
```

The following snippet of code demonstrates some of the ways to create a datagram packet.

```
// Create a packet to receive 1024 bytes of data
byte[] data = new byte[1024];
DatagramPacket packet = new DatagramPacket(data, data.length);

/* Create a packet that a has buffer size of 1024, but it will receive
   data starting at offset 8 (offset zero means the first element
   in the array) and it will receive only 32 bytes of data.
*/
byte[] data2 = new byte[1024];
DatagramPacket packet2 = new DatagramPacket(data2, 8, 32);

/* Create a packet to send 1024 bytes of data that has
   A destination address of "localhost" and port 15900. Will need to
   populate data3 array before sending the packet.
*/
byte[] data3 = new byte[1024];
DatagramPacket packet3 = new DatagramPacket(data3, 1024,
                            InetAddress.getByName("localhost"), 15900);

/* Create a packet to send 1024 bytes of data that has
   A destination address of "localhost" and port 15900. Will need to
   populate data4 array before sending the packet. The code sets the
   destination address by calling methods on the packet instead of
   specifying it in its constructor.
*/
```

```
byte[] data4 = new byte[1024];
DatagramPacket packet4 = new DatagramPacket(data4, 1024);
packet4.setAddress(InetAddress.getByName("localhost"));
packet4.setPort(15900);
```

It is very important to understand that data in the packet always has offset and length specified. You need to use those two pieces of information while reading data from a packet. Suppose that a `receivedPacket` object reference represents a `DatagramPacket` that you have received from a remote UDP socket. The `getData()` method of the `DatagramPacket` class returns the buffer (a byte array) of the packet. A packet can have a bigger buffer than the size of the received data from a remote client. In such cases, you must use an offset and a length to grab the data from the buffer that was received without touching the garbage data in the buffer. If a packet's buffer size is smaller than the size of the data received, the extra bytes are silently ignored. You should use the code similar to the one shown below to grab data that a socket receives. The point is that you should use data in the receiving buffer starting from its specified `offset` and as many bytes as indicated by its `length` property.

```
// Get the packet's buffer, offset and length
byte[] dataBuffer = receivedPacket.getData();
int offset = receivedPacket.getOffset();
int length = receivedPacket.getLength();

// Copy the received data using offset and length to receivedData
// array, which will hold all good data
byte[] receivedData = new byte[length];
System.arraycopy(dataBuffer, offset, receivedData, 0, length);
```

Creating a UDP socket (client as well as server) is as simple as creating an object of the `DatagramSocket` class. You can use its `send()` method to send a packet to its destination. You can use the `receive()` method to receive a packet from a remote socket. The `receive()` method blocks until a packet arrives. You supply an empty datagram packet to the `receive()` method. The socket populates it with information that it receives from the remote socket. If the supplied datagram packet has a smaller data buffer size than that of the received datagram packet, the received data is truncated silently to fit into the supplied datagram packet. If the supplied datagram packet has a bigger data buffer size than that of the received one, the socket will copy the received data to the supplied data buffer in its segment indicated by its offset and length properties and would not touch the other parts of the buffer. Note that the available data buffer size is not the size of the byte array. Rather, it is defined by the length. For example, suppose you have a datagram packet with a byte array of 32 elements with an offset of 2 and a data buffer length of 8. If you pass this datagram packet to the `receive()` method, the maximum of 8 bytes of received data will be copied. The data will be copied from the 3rd element in the buffer to the 11th element as indicated by the offset 2 and the length 8 respectively.

```
// Create a UDP socket bound to a port number 15900 at localhost
DatagramSocket socket = new DatagramSocket(15900,
                            InetAddress.getByName("localhost"));

// Send a packet assuming that you have a datagram packet in p
socket.send(p);

// Receive a packet
DatagramPacket p2 = new DatagramPacket(new byte[1024], 1024);
socket.receive(p2);
```

Creating a UDP Echo Server

Creating an echo server using UDP is very easy. It takes only four lines of real code.

- Create a `DatagramSocket` object to represent a UDP socket.
- Create a `DatagramPacket` object to receive the packet from a remote client.
- Call the `receive()` method of the socket to wait for a packet to arrive. .
- Call the `send()` method of the socket passing the same packet that you received. When a UDP packet is received by a server, it contains the sender's address. You do not need to change anything in the packet to echo back the same message to the sender of the packet. When you prepare a datagram packet for sending, you need to set a destination address. When the packet arrives at its destination, it contains its sender's address. This is useful in case the receiver wants to respond to the sender of the datagram packet.

The following snippet of code shows how to write a UDP echo server.

```
DatagramSocket socket = new DatagramSocket(15900);
DatagramPacket packet = new DatagramPacket(new byte[1024], 1024);
while(true) {
     // Receive the packet
     socket.receive(packet);

     //Send back the same packet to the sender
     socket.send(packet);
}
```

Listing 3-5 has the expanded version of the same code for a UDP echo server. It contains the same basic logic as shown above. Additionally, it has code to handle errors and print the packet's details on the standard output.

Listing 3-6 is for a client application that uses a UDP socket to send/receive messages to/from the UDP echo server. Note that the client and server exchange one line of text at a time.

To test the UDP echo application, you need to run `UDPEchoServer` and `UDPEchoClient` classes. You need to run the server first. The client application will prompt you to enter a message. Enter a text message and press the Enter key to send that message to the server. The server will echo back the same message. Both applications display the messages being exchanged on the standard output. They also display the packet details, such as the sender's IP address and port number. The server application uses a port number 15900 and the client application uses any available UDP port on the computer. If you get an error, it means that port number 15900 is in use, and you will need to change the port number in the server program and use the new port number in the client program to address the packet. The following text shows the logs on the client and server consoles for a typical interaction between these two applications. The server is designed to handle multiple clients at a time. You can run multiple instances of the `UDPEchoClient` class. Note that the server runs in an infinite loop and you must stop the server application manually,

```
Console log on the server

Created UDP server socket at /127.0.0.1:15900...
Waiting for a UDP packet to arrive...
Received a packet:[IP Address=/127.0.0.1, port=1522, message=Hello]
Waiting for a UDP packet to arrive...
Received a packet:[IP Address=/127.0.0.1, port=1522, message=Nice talking
```

```
to you]
Waiting for a UDP packet to arrive...


Console log on the client:

Started UDP client socket at 0.0.0.0/0.0.0.0:1522
Please enter a message (Bye to quit):Hello
[Server at IP Address=localhost/127.0.0.1, port=15900]: Hello

Please enter a message (Bye to quit):Nice talking to you
[Server at IP Address=localhost/127.0.0.1, port=15900]: Nice talking to
you

Please enter a message (Bye to quit):bye
```

Listing 3-5: An echo server based on a UDP sockets

```
// UDPEchoServer.java
package com.jdojo.chapter3;

import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class UDPEchoServer {
     public static void main(String[] args) {
          final int LOCAL_PORT = 15900;
          final String SERVER_NAME = "localhost";

          try {
               DatagramSocket udpSocket = new DatagramSocket(LOCAL_PORT,
                                    InetAddress.getByName(SERVER_NAME));

               System.out.println("Created UDP server socket at " +
                                  udpSocket.getLocalSocketAddress() +
                                  "...");

               // Wait for a message in a loop and echo the same
               // message to the sender
               while (true) {
                    System.out.println("Waiting for a UDP packet" +
                                       " to arrive...");

                    // Prepare a packet to hold the received data
                    DatagramPacket packet =
                              new DatagramPacket(new byte[1024], 1024);

                    // Receive a packet
                    udpSocket.receive(packet);

                    // Print the packet details
                    displayPacketDetails(packet);

                    // Echo the same packet to the sender
                    udpSocket.send(packet);
```

```
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void displayPacketDetails(DatagramPacket packet) {
        // Get the message
        byte[] msgBuffer = packet.getData();
        int length = packet.getLength();
        int offset = packet.getOffset();

        int remotePort = packet.getPort();
        InetAddress remoteAddr = packet.getAddress();
        String msg = new String(msgBuffer, offset, length);

        System.out.println("Received a packet:[IP Address=" +
                           remoteAddr +
                           ", port=" + remotePort +
                           ", message=" + msg + "]");
    }
}
```

Listing 3-6: An echo client based on a UDP socket

```
// UDPEchoClient.java
package com.jdojo.chapter3;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.UnknownHostException;

public class UDPEchoClient {
    public static void main(String[] args) {
        DatagramSocket udpSocket = null;
        BufferedReader br = null;
        try {
            // Create a UDP socket at local host using any
            // available port
            udpSocket = new DatagramSocket();

            String msg = null;

            // Create a buffered reader to get an input from a user
            br = new BufferedReader(new InputStreamReader(System.in));

            String promptMsg = "Please enter a message (Bye to quit):";
            System.out.print(promptMsg);

            while ((msg = br.readLine()) != null) {
                if (msg.equalsIgnoreCase("bye")) {
                    break;
                }
```

```
                // Prepare a packet to send to the server
                DatagramPacket packet = UDPEchoClient.getPacket(msg);

                // Send the packet to the server
                udpSocket.send(packet);

                // Wait for a packet from the server
                udpSocket.receive(packet);

                // Display the packet details received from the server
                displayPacketDetails(packet);

                System.out.print(promptMsg);
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            // Close the socket
            if (udpSocket != null) {
                udpSocket.close();
            }
        }
    }

    public static void displayPacketDetails(DatagramPacket packet) {
        byte[] msgBuffer = packet.getData();
        int length = packet.getLength();
        int offset = packet.getOffset();
        int remotePort = packet.getPort();
        InetAddress remoteAddr = packet.getAddress();
        String msg = new String(msgBuffer, offset, length);
        System.out.println("[Server at IP Address=" + remoteAddr +
                           ", port=" + remotePort + "]: " + msg);

        // Add a line break
        System.out.println();
    }

    public static DatagramPacket getPacket(String msg)
                                        throws UnknownHostException {
        // We will send and accept a message of 1024 bytes in length.
        // longer messages will be truncated
        final int PACKET_MAX_LENGTH = 1024;
        byte[] msgBuffer = msg.getBytes();

        int length = msgBuffer.length;
        if (length > PACKET_MAX_LENGTH) {
            length = PACKET_MAX_LENGTH;
        }

        DatagramPacket packet = new DatagramPacket(msgBuffer, length);

        // Set the destination address and the port number
        int serverPort = 15900;
```

```
            final String SERVER_NAME = "localhost";
            InetAddress serverIPAddress =
                      InetAddress.getByName(SERVER_NAME);
            packet.setAddress(serverIPAddress);
            packet.setPort(serverPort);

            return packet;
        }
}
```

A Connected UDP Socket

UDP is a connectionless protocol. UDP sockets do not support an end-to-end connection like the TCP sockets. However, the `DatagramSocket` class has a `connect()` method. This method allows an application to restrict sending and receiving of UDP packets to a specific IP address at a specific port number. Let us consider the following snippet of code.

```
InetAddress localIPAddress = InetAddress.getByName("192.168.11.101");
int localPort = 15900;
DatagramSocket socket = new DatagramSocket(localPort, localIPAddress);

// Connect the socket to a remote address
InetAddress remoteIPAddress = InetAddress.getByName("192.168.12.115");
int remotePort = 17901;
socket.connect(remoteIPAddress, remotePort);
```

The socket is bound to the local IP address `192.168.11.101` and local UDP port number `15900`. It is connected to a remote IP address of `192.168.12.115` and a remote UDP port number `17901`. It means that the `socket` object can be used to send/receive a datagram packet only to/from another UDP socket running at an IP address of `192.168.12.115` at a port number `17901`. After you have called the `connect()` method on a UDP socket, you do not need to set the destination IP address and the port number for the outgoing datagram packets. The socket will add the destination IP address and port number that were used in the `connect()` method's call, to all outgoing packets. If you do supply a destination address with a packet before you send it, the socket will make sure the destination address supplied in the packet is the same as the remote address used in the `connect()` method call. If the destination address supplied in a packet and the remote address used to connect the socket do not match, the `send()` method will throw an exception.

You may realize that using the `connect()` method of a UDP socket has two advantages:

- It sets the destination address for the outgoing packets every time you send a packet.
- It restricts the socket to communicate only to the remote host whose IP address was used in the `connect()` method's call.

Now, we understand that UDP sockets are connectionless and we do not have a real connection using a UDP socket. The `connect()` method in the `DatagramSocket` class does not provide any kind of connection for UDP sockets. Rather, it is useful for restricting the communication to a specific remote UDP socket.

UDP Multicast Sockets

Java supports UDP multicast sockets, which can receive datagram packets sent to a multicast IP address. An object of the `MulticastSocket` class represents a multicast socket. Working with a `MulticastSocket` socket is similar to working with a `DatagramSocket` with one difference. A multicast socket is based on a group membership. After you have created and bound a multicast socket, you will need to call its `joinGroup(InetAddress multiCastIPAddress)` method to make this socket a member of the multicast group defined by the specified multicast IP address, `multiCastIpAddress`. Once it becomes a member of a multicast group, any datagram packet sent to that group will be delivered to this socket. There can be multiple members in a multicast group. A multicast socket can be a member of multiple multicast groups. If a member decides not to receive a multicast packet from a group, it can leave the group by calling its `leaveGroup(InetAddress multiCastIPAddress)` method.

In IPv4, any IP address in the range `224.0.0.0` to `239.255.255.255`, can be used as a multicast address to send a datagram packet. The IP address `224.0.0.0` is reserved and you should not use it in your application. A multicast IP address cannot be used as a source address for a datagram packet, which implies that you cannot bind a socket to a multicast address.

A socket itself does not have to be a member of a multicast group to send a datagram packet to a multicast address.

TIP
Java 7 has added the IP multicast capability to the `DatagramChannel` class. Please refer to the *Multicasting Using Datagram Channels* section later in this chapter on how to use a datagram channel for IP multicasting. Note that the `DatagramChannel` class was added in Java 1.4, which did not have the IP multicast capability.

Listing 3-7 has a program that creates a multicast socket that receives datagram packets addressed to the 230.1.1.1 multicast IP address.

Listing 3-8 has a program that sends a message to the same multicast address. Note that you can run multiple instances of the `UDPMulticastReceiver` class and all of them will become a member of the same multicast group. When you run the `UDPMulticastSender` class, it will send a message to the group and all members in the group will receive a copy of the same message. Note that the sender of a multicast message does not have to be a member of that multicast group. The `UDPMulticastSender` class uses a `DatagramSocket` and not a `MulticastSocket` to send a multicast message.

Listing 3-7: A UDP multicast socket that receives UDP multicast messages

```
// UDPMultiCastReceiver.java
package com.jdojo.chapter3;

import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;

public class UDPMultiCastReceiver {
     public static void main(String[] args) {
          int mcPort = 18777;
          String mcIPStr = "230.1.1.1";
```

```
          MulticastSocket mcSocket = null;
          InetAddress mcIPAddress = null;
          try {
               mcIPAddress = InetAddress.getByName(mcIPStr);
               mcSocket = new MulticastSocket(mcPort);
               System.out.println("Multicast Receiver running at:" +
                                  mcSocket.getLocalSocketAddress());

               // Join the group
               mcSocket.joinGroup(mcIPAddress);

               DatagramPacket packet =
                         new DatagramPacket(new byte[1024], 1024);

               while (true) {
                    System.out.println("Waiting for a multicast" +
                                       " message...");

                    mcSocket.receive(packet);

                    String msg = new String(packet.getData(),
                                            packet.getOffset(),
                                            packet.getLength());
                    System.out.println("[Multicast Receiver] Received:" +
                                       msg);
               }
          }
          catch (Exception e) {
               e.printStackTrace();
          }
          finally {
               if (mcSocket != null) {
                    try {
                         mcSocket.leaveGroup(mcIPAddress);
                         mcSocket.close();
                    }
                    catch (Exception e) {
                         e.printStackTrace();
                    }
               }
          }
     }
}
```

Listing 3-8: A UDP datagram socket - a multicast sender application

```
// UDPMultiCastSender.java
package com.jdojo.chapter3;

import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class UDPMulticastSender {
     public static void main(String[] args) {
          int mcPort = 18777;
          String mcIPStr = "230.1.1.1";
```

```
        DatagramSocket udpSocket = null;

        try {
             // Create a datagram socket
             udpSocket = new DatagramSocket();

             // Prepare a message
             InetAddress mcIPAddress = InetAddress.getByName(mcIPStr);

             byte[] msg = "Hello multicast socket".getBytes();
             DatagramPacket packet =
                                 new DatagramPacket(msg, msg.length);
             packet.setAddress(mcIPAddress);
             packet.setPort(mcPort);
             udpSocket.send(packet);

             System.out.println("Sent a multicast message.");
             System.out.println("Exiting application");
        }
        catch (Exception e) {
             e.printStackTrace();
        }
        finally {
             if (udpSocket != null) {
                  try {
                       udpSocket.close();
                  }
                  catch (Exception e) {
                       e.printStackTrace();
                  }
             }
        }
    }
}
```

URI, URL and URN

A Uniform Resource Identifier (URI) is a sequence of characters that identifies a resource. The Request for Comments (RFC) 3986 defines the generic syntax for URI. Full text of this RFC is available at *http://www.ietf.org/rfc/rfc3986.txt*. A resource identifier can identify a resource by a location, a name, or both. This section gives an overview of URI. If you are interested in details about URI, you are advised to read RFC3986.

A URI that uses a location to identify a resource is called Uniform Resource Locator (URL). For example, http://www.yahoo.com/index.html represents a URL that identifies a document named index.html at a host www.yahoo.com. mailto:kishori.sharan@jdojo.com is another example of a URL. In this URL, the mailto protocol instructs the application, which interprets it, to open up an email application to send an email to the email address specified in the URL. In this case, a URL is not locating any resources. Rather, it is identifying the details of an email. You can also set the subject and the body parts of an email using the mailto protocol. Therefore, a URL does not always imply a location of a resource. Sometimes, the resource may be abstract as in the case of a mailto protocol. Once you locate a resource using a URL, you can perform some operations, such as retrieve, update, or delete, on the resource. The details of how the operations are performed depend on the scheme being used in a URL. A URL just identifies the parts of a resource location

and scheme to locate it and not the details of any operations that can be performed on the resource.

A URI that uses a name to identify a resource is called a Uniform Resource Name (URN). For examples, URN:ISBN:0-395-36341-1 `represents a URN, which identifies a book using` International Standard Book Numbers (ISBN) namespace.

URL and URN are subsets of URI. Therefore, the discussion about URI applies to both URL and URN. The detailed syntax of a URI depends on the scheme it uses. In this section, we will cover a generic syntax of a URI, which is typically a URL. The next section will explore Java classes that are used to represent URI and URL in a Java program.

A URI can be absolute or relative. A relative URI is always interpreted in the context of another absolute URI, which is called a base URI. In other words, you must have an absolute URI to make a relative URI meaningful. An absolute URI has the following generic format.

```
<scheme>:<scheme-specific-part>
```

The `<scheme-specific-part>` syntax details depend on the `<scheme>` value. For example, a `http` scheme uses one format, and a `mailto` scheme uses another format. Another generic form of a URI is as follows. Typically, but not necessarily, it represents a URL.

```
<scheme>://<authority><path>?<query>#<fragment>
```

Here, `<scheme>` indicates a method to accessing a resource. It is the protocol name such as http, ftp, etc. We all use the term “protocol” for what is termed a “scheme” in the URI specification. If the term scheme throws you off, you can read it as “protocol” whenever it appears in this section. The `<scheme>` and `<path>` parts are required in a URI. All other parts are optional. The `<path>` part may be an empty string.

The `<authority>` part indicates the server name (or IP address) or a scheme-specific registry. If the `<authority>` part represents a server name, it may be written in the following form of `<userinfo>@host:port`. If a `<authority>` is present in a URI, it begins with two forward slashes. It is an optional part. For example, a URL that identifies a file in a local file system on a machine uses *file* scheme as file:/c:/documents/welcome.doc.

URI syntax uses a hierarchical syntax in its `<path>` part, which locates the resource on the server. Multiple parts of the `<path>` are separated by a forward slash (`/`).

The `<query>` part indicates that the resource is obtained by executing the query. It consists of name-value pairs separated by an ampersand (&). The name and value are separated by an equal sign (=). For example, `id=123&rate=5.5` is a query, which has two parts, `id` and `rate`. The value for `id` is `123` and the value for `rate` is 5.5.

The `<fragment>` part identifies a secondary resource, typically a subset of the primary resource identified by another part of the URI.

The following is an example of a URI, which is also broken into parts. It is a URL that refers to a document named intro.html on www.jdojo.com server. The scheme, http, indicates that the document can be retrieved using an http protocol. The query part, id=123, indicates that the document is obtained by executing this query. The fragment part, conclusion, can be interpreted differently by the application that uses the document. In case of an HTML document, the fragment part is interpreted by a web browser as a part of the main document.

```
URI:       http://www.jdojo.com/java/intro.html?id=123#conclusion
Scheme:    http
Authority: www.jdojo.com
Path:      /java/intro.html
Query:     id=123
Fragment:  conclusion
```

Not all parts of a URI are mandatory. Which parts are mandatory and which parts are optional depend on the scheme that is used. One of the goals of using a URI to identify a resource was to make it universally readable. For this reason, there is a well-defined set of characters that can be used to represent a URI. URI syntax uses some reserved characters that have special meaning and they can only be used in specific parts of a URI. In other parts, the reserved characters need to be escaped. A character is escaped by using a percent character followed by its ASCII value in a hexadecimal format. For example, ASCII value of space is 32 in a decimal format, and it is 20 in a hexadecimal format. If you want to use a space character in a URI, you must use %20, which is the escaped form for a space. Since the percent sign is used as part of an escape character, you must use %25 to represent a % character in a URI (25 is the hexadecimal value for number 37 in decimal. The ASCII value for % is 37 in decimal). For example, if you want to use a value of 5.2% in a query part, the following is an invalid URI.

```
http://www.jdojo.com/details?rate=5.2%
```

You need to escape the percent sign character as %25 as shown below.

```
http://www.jdojo.com/details?rate=5.2%25
```

It is important to understand the usage of a relative URI. A relative URI is always interpreted in the context of an absolute URI, which is called the based URI. An absolute URI starts with a scheme. A relative URI inherits some parts of its base URI. Let us consider a URI that refers to an HTML document as shown below.

```
http://www.jdojo.com/java/intro.html
```

The document referred to in the URI is intro.html. Its path is /java/intro.html. Suppose two documents, brief_intro.html and detailed_intro.html, reside (physically or logically) in the same path hierarchy as intro.html. The following are the absolute URI for all three documents.

```
http://www.jdojo.com/java/intro.html
http://www.jdojo.com/java/brief_intro.html
http://www.jdojo.com/java/detailed_intro.html
```

If we are already in the intro.html context, it will be easier to refer to the other two documents using their names instead of their absolute URI. What do we mean by being in the intro.html context? When we use a *http://www.jdojo.com/java/intro.html* URI to identify a resource, it has three parts – a scheme (http), a server name (*www.jdojo.com*), and a document path (/java/introl.html). The path indicates that the document is under the *java* path hierarchy, which in turn, is at the root of the path hierarchy. All details – scheme, server name, path details, excluding the document name itself (intro.html) make up the context for the intro.html document. If you look at the URI for the other two documents listed above, you will notice that all details about them are the same as for intro.html. In other words, we can state that the context for the other two documents is the same as for intro.html. In this case, with an absolute URI of the intro.html document as base URI, the relative URI for the other two documents would be their names – brief_introl.html and detailed_intro.html. It can be listed as:

```
Base URI: http://www.jdojo.com/java/intro.html
Relative URI: brief_intro.html
Relative URI: detailed_intro.html
```

In the above list, the two relative URIs inherit the scheme, server name and path hierarchy from the base URI. It is to be emphasized that a relative URI never makes sense without specifying its base URI.

When a relative URI has to be used, it must be resolved to its equivalent absolute URI. The URI specification lays down rules to resolve a relative URI. We will discuss some of the most commonly used forms of relative URIs and their resolutions. There are two special characters used to define the `<path>` part of a URI. They are `.` (a dot) and `..` (two dots). A dot means the current path hierarchy. Two dots mean one up in the path hierarchy. You must have seen these two sets of characters being used in a file system to mean the current folder and parent folder. You can think of their meanings in a URI the same way, but a URI does not assume any folder hierarchy. In a URI, a path is considered as hierarchical, and it is not tied to a file system hierarchical structure at all. However, in practice, when you work with web-based applications, URLs are usually mapped to a file system hierarchical structure. In the normalized form of a URI, dots are replaced appropriately. For example, `s://sn/a/./b` is normalized to `s://sn/a/b`, and `s://sn/a/../b` is normalized to `s://sn/b`. The non-normalized and normalized forms refer to the same URL. The normalized form has extra characters removed. By just looking at two URIs, you cannot say that they are referring to the same resource or not. You must normalize them before you compare them for equality. During the comparison process, scheme, server name, and hexadecimal digits are considered case-insensitive.

Here are some rules to resolve a relative URI. Table 3-5 has examples of all rules listed below.

- If a URI starts with a scheme, it is considered an absolute URI.
- If a relative URI starts with an authority, it inherits scheme from its base URI.
- If a relative URI is an empty string, it is the same as the base URI.
- If a relative URI has a fragment part only, the resolved URI uses the new fragment. If a base URI had a fragment, it would be replaced with the fragment of the relative URI. Otherwise, the fragment of the relative URI is added to the base URI.
- Relative URI's path does not start with a forward slash (`/`). If the base URI has a path, remove the last component of the path in the base URI and append the relative URI. Note that the last component of the path may be an empty string as in *http://www.abc.com/*.
- If a relative URL starts with a path, which in turn starts with a forward slash (`/`), the base URI's path is replaced with the relative URI's path.

Table 3-5: Examples of how a relative URI is resolved to an absolute URI using its base URI. The examples in this table conform to the rules followed in Java URI and URL classes. Java rules deviate slightly in a few cases from the rules set in the URI specification.

Comments about Relative URI	Base URI	Relative URI	Resolved Relative URI
Absolute URI	`h://sn/a/b/c`	`http://sn2/foo`	`h://sn2/foo`
Starts with authority	`h://sn/a/b/c`	`//sn2/h/k`	`h://sn2/h/k`
Empty string	`h://sn/a/b/c`		`h://sn/a/b/c`
Has fragment only	`h://sn/a/b/c`	`#k`	`h://sn/a/b/c#k`
	`h://sn/a/b/c#a`	`#k`	`h://sn/a/b/c#k`
Path does not start with	`h://sn/a/b/`	`foo`	`h://sn/a/b/foo`

a forward slash (/)	h://sn/a/b/c	foo	h://sn/a/b/foo
	h://sn/a/b/c?d=3	foo	h://sn/a/b/foo
	h://sn/	foo	h://sn/foo
	h://sn	foo	h://sn/foo
Path starts with a forward slash (/)	h://sn/a/b/	/foo	h://sn/foo
	h://sn/a/b/c	/foo	h://sn/foo
	h://sn/a/b/c?d=3	/foo	h://sn/foo
	h://sn/	/foo	h://sn/foo
	h://sn	/foo	h://sn/foo

TIP
You can also use a host name or IP address as an authority in a URI. IPv4 can be used in its dotted decimal format, e.g., http://192.168.10.178/docs/toc.html. IPv6 must be enclosed in brackets, e.g., http://[1283::8:800:200C:A43A]/docs/toc.html.

URI and URL as Java Objects

Java represents a URI and a URL as objects. It provides the following four classes that you can use to work with a URI and a URL as objects in a Java program.

- `java.net.URI`
- `java.net.URL`
- `java.net.URLEncoder`
- `java.net.URLDecoder`

An object of the `URI` class represents a URI. An object of the `URL` class is used to represent a URL. `URLEncoder` and `URLDecoder` are utility classes that help encode and decode URI strings. We will discuss other Java classes in the next sections that are used to retrieve the resource identified by a URL.

The `URI` class has many constructors. They let you pass variable combinations of parts (scheme, authority, path, query and fragment) of a URI. All constructors throw a checked exception - `URISyntaxException`. This will force you to handle the exception in your code when you create a URI object. Exception is thrown because of strings, which you use to construct a `URI` object, may not be in conformity with the URI specification.

```
// Create a URI object
URI baseURI = new URI("http://www.yahoo.com");

// Create a URI with relative URI string and resolve it using baseURI
URI relativeURI = new URI("welcome.html");
URI resolvedRelativeURI = baseURI.resolve(relativeURI);
```

Listing 3-9 demonstrates how to use the `URI` class in a Java program. You can also get a `URL` object from a `URI` object using its `toURL()` method as shown below.

```
URL baseURL = baseURI.toURL();
```

You can also create a `URI` object using the `create(String str) static` method of the URI class. The `create()` method does not throw a checked exception. It throws a runtime exception. Therefore, its use will not force you to handle the exception. You should use this method only when you know that a URI string is well-formed.

```
URI uri2 = URI.create("http://www.yahoo.com");
```

Listing 3-9: A sample class that demonstrates the use of the java.net.URI class

```
// URITest.java
package com.jdojo.chapter3;

import java.net.URI;
import java.net.URISyntaxException;

public class URITest {
     public static void main(String[] args) {
          String baseURIStr = "http://www.jdojo.com/javaintro.html?" +
                              "id=25&rate=5.5%25#foo";
          String relativeURIStr = "../sports/welcome.html";

          try {
               URI baseURI = new URI(baseURIStr);
               URI relativeURI = new URI(relativeURIStr);

               // Resolve the relative URI with respect to the base URI
               URI resolvedURI = baseURI.resolve(relativeURI);

               printURIDetails(baseURI);
               printURIDetails(relativeURI);
               printURIDetails(resolvedURI);
          }
          catch (URISyntaxException e) {
               e.printStackTrace();
          }
     }

     public static void printURIDetails(URI uri) {
          System.out.println("URI:" + uri);
          System.out.println("Normalized:" + uri.normalize());
          String parts = "[Scheme=" + uri.getScheme() +
                         ", Authority=" + uri.getAuthority() +
                         ", Path=" + uri.getPath() +
                         ", Query:" + uri.getQuery() +
                         ", Fragment:" + uri.getFragment() + "]";

          System.out.println(parts);
          System.out.println();
     }
}
```

```
Output:

RI:http://www.jdojo.com/javaintro.html?id=25&rate=5.5%25#foo
```

```
Normalized:http://www.jdojo.com/javaintro.html?id=25&rate=5.5%25#foo
[Scheme=http, Authority=www.jdojo.com, Path=/javaintro.html,
Query:id=25&rate=5.5%, Fragment:foo]

URI:../sports/welcome.html
Normalized:../sports/welcome.html
[Scheme=null, Authority=null, Path=../sports/welcome.html, Query:null,
Fragment:null]

URI:http://www.jdojo.com/../sports/welcome.html
Normalized:http://www.jdojo.com/../sports/welcome.html
[Scheme=http, Authority=www.jdojo.com, Path=/../sports/welcome.html,
Query:null, Fragment:null]
```

An instance of the `java.net.URL` class represents a URL in a Java program. Although every URL is also a URI, Java does not inherit the `URL` class from the `URI` class. Java uses the term protocol to refer to the scheme part in the URI specification. You can create a URL object by providing a string that has all URL's parts concatenated, or by providing parts of a URL separately. If strings that you supply to create a URL object are not valid, the constructors of the URL class will throw a `MalformedURLException` checked exception. You must handle this exception when you create a `URL` object. Listing 3-10 demonstrates how to create a `URL` object. The `URL` class lets you create an absolute URL from a relative URL and a base URL using one of its constructors.

Listing 3-10: A sample class that demonstrates the use of the java.net.URL class

```
// URLTest.java
package com.jdojo.chapter3;

import java.net.URL;

public class URLTest {
     public static void main(String[] args) {
          String baseURLStr = "http://www.ietf.org/rfc/rfc3986.txt";
          String relativeURLStr = "rfc2732.txt";
          try {
               URL baseURL = new URL (baseURLStr);
               URL resolvedRelativeURL = new URL(baseURL, relativeURLStr);
               System.out.println("Base URL:" + baseURL);
               System.out.println("Relative URL String:" +
                                  relativeURLStr);
               System.out.println("Resolved Relative URL:" +
                                  resolvedRelativeURL);
          }
          catch (Exception e) {
               e.printStackTrace();
          }
     }
}
```

```
Output:

Base URL:http://www.ietf.org/rfc/rfc3986.txt
Relative URL String:rfc2732.txt
Resolved Relative URL:http://www.ietf.org/rfc/rfc2732.txt
```

Typically, you create a `URL` object to retrieve the resource identified by the URL. Note that you can create an object of the `URL` class as long as the URL is well formed textually and the protocol to handle the URL is available. The successful creation of a `URL` object in a Java program does not guarantee the existence of the resource at the server specified in the URL. The `URL` class provides methods that you can use in conjunction with other classes to retrieve the resource identified by a URL.

The `URL` class makes sure that it can handle the protocol specified in the URL string. For example, it will not let you create a `URL` object with a string as ppp://www.sss.com/ unless you develop and supply it a protocol handler for a ppp protocol. We will discuss details on how to retrieve the resource identified by a URL in the next section.

Sometimes, you do not know the parts of the URL string in advance. You get the parts of the URL at runtime as input from other parts of the program, or from the user. In such cases, you will need to encode the parts of the URL before you can use them to create a `URL` object. Sometimes, you get a string in encoded form and you want it to be decoded. An encoded string will have all the restricted characters properly escaped.

The `URLEncoder` and `URLDecoder` classes are used to encode and decode strings respectively. The `URLEncoder.encode(String source, String encoding) static` method is used to encode a source string using the specified encoding. The `URLDecoder.decode(String source, String encoding) static` method is used to decode a source string using a specified encoding. The following snippet of code shows how to encode/decode strings. Typically, you encode/decode the value part of name-value pairs in the query part of a URL. Note that you should never attempt to encode the entire URL string. Otherwise, it will encode some of the reserved characters such a forward slash and the resulting URL string will be invalid.

```
String source = "this is a test for 2.5% and &" ;
String encoded = URLEncoder.encode(source, "utf-8");
String decoded = URLDecoder.decode(encoded, "utf-8");
System.out.println("Source: " + source);
System.out.println("Encoded: " + encoded);
System.out.println("Decoded: " + decoded);
```

```
Output:

Source: this is a test for 2.5% and &
Encoded: this+is+a+test+for+2.5%25+and+%26
Decoded: this is a test for 2.5% and &
```

Accessing the Contents of a URL

`A` URL has a protocol that is used to communicate with the remote application that hosts the URL's contents. For example, the *http://www.yahoo.com/index.html* URL uses a `http` protocol. In a URL, we specify a protocol that is used by the application layer in the protocol suite. When we need to access a URL's contents, the computer will use some kind of protocols from lower layers in the protocol suite (transport, Internet layers, etc.) to communicate with the remote host. The `http` application layer protocol uses `TCP/IP` protocols in lower layers. In a distributed application, it is very frequent that you need to retrieve (or read) the resource (could be text, html contents, image files, audio/video files or any other kind of information) identified by a URL. Although it is possible to open a socket every time you need to read the contents of URL, it is time consuming and

cumbersome for programmers. After all, programmers need some way to be more productive than writing repetitive code for what seems to be a routine job. Java designers realized this need and they have provided a very easy (yes, it is very easy) way to read/write data from/to a URL. This section will explore some of the ways - from a very simple to complex, to read/write data from/to a URL.

If you recall that, as the data passes from one layer to another in the protocol suite, each layer adds a header to the data. Since a URL uses a protocol in the application layer, it also contains its own header. The format of the header depends on the protocol being used. When an http request is send to a remote host, the application layer in the source host adds an http header to the data. The remote host has an application layer that handles http protocol and it uses the header information to interpret the contents. In summary, a URL data will have two parts – a header part and a contents part. The `URL` class along with some other classes let you read/write both header and content parts of a URL. We will start with the simplest case of reading the contents of a URL.

Before we read/write from/to a URL, we need to have a working URL, which we can access. You can read content of any URL that is publicly available on Internet. For our discussion purpose, we assume that you are familiar with Java Server Pages (JSP) and you have access to a web server, where you can deploy a JSP page. If you do not know JSP, you can just replace the URL used in examples of this section with any publicly available URL, for example, *http://www.yahoo.com* will work fine, and you should be able to run all examples. Writing data to a URL is a little different. It will be easier if you can run your JSP to see how writing to a URL works. We assume that you have deployed a web application on a web server and it has a web page called *echo_params.jsp*.

Listing 3-11 shows the content of this JSP page. It performs two things. It reads the `HTTP` request method, which can be GET or POST, and prints it. It reads all the parameters passed in with the `HTTP` request and prints the list of parameter names and values.

Listing 3-11: The content of the echo_params.jsp file

```
<%@ page contentType="text/html;charset=windows-1252"%>
<html>
     <head>
          <meta http-equiv="Content-Type" content="text/html;
               charset=windows-1252"/>
          <title>Echo Request Method and Parameters</title>
     </head>
     <body>
          <h1>URL Connection Test</h1>
          <%
          out.println("Request Method: " + request.getMethod());
          out.println("<br/><br/>");

          out.println("<u>List of Parameter Names and Values</u><br/>");
          java.util.Enumeration paramNames =
               request.getParameterNames();
          while(paramNames.hasMoreElements()) {
               String paramName  = (String)paramNames.nextElement();
               String paramValue = request.getParameter(paramName);
               out.println("Name: " + paramName + ", Value: " +
                           paramValue);
               out.println("<br/>");
          }
          %>
     </body>
</html>
```

The `URL` class lets you read the contents (not header) of a URL by just writing two lines of code as shown below.

```
URL url = new URL("your URL string goes here");
InputStream ins = url.openStream();
```

Listing 3-12 has the complete program that reads a URL's contents. You will need to change the URL in this program accordingly to your web server setup. The output shows that we do access the JSP and JSP gets the query (`id=123`) passed to it and transmits back the generated HTML contents. The HTML request was sent using the GET method. If you want to use the POST method to send a request to a URL, you will need to use the `URLConnection` class, which we will discuss next.

Listing 3-12: A simple URL content reader program

```
// SimpleURLContentRetrieval
package com.jdojo.chapter3;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;

public class SimpleURLContentReader {
     public static String getURLContent(String urlStr) {
          BufferedReader br = null;
          try {
               URL url = new URL(urlStr);

               // Get the input stream
               InputStream ins = url.openStream();

               // Wrap input stream into a reader
               br = new BufferedReader(new InputStreamReader(ins));

               StringBuffer sb = new StringBuffer();
               String msg = null;
               while ((msg = br.readLine()) != null) {
                    sb.append(msg);
                    sb.append("\n"); // Append a new line
               }

               return sb.toString();
          }
          catch (Exception e) {
               e.printStackTrace();
          }
          finally {
               if (br != null) {
                    try {
                         br.close();
                    }
                    catch (IOException e) {
                         e.printStackTrace();
                    }
```

```
            }
        }

        // If we get here it means there was an error
        return null;
    }

    public static void main(String[] args) {
        String urlStr = "http://localhost:8080/docsapp/" +
                        "echo_params.jsp?id=123";
        String content = getURLContent(urlStr);
        System.out.println(content);
    }
}
```

Output: (The output has been reformatted for readability.)

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
        charset=windows-1252"/>
    <title>Echo Request Method and Parameters</title>
  </head>
  <body>
    <h1>URL Connection Test</h1>
    Request Method: GET
    <br/><br/>
    <u>List of Parameter Names and Values</u><br/>
    Name: id, Value: 123
   <br/>
 </body>
</html>
```

Once you get the input stream, you can use it for reading the content of the URL. Another way of reading the content of a URL is by using the `getContent()` method of the `URL` class. Since `getContent()` can return any kind of content, its return type is the `Object` type. You will need to check what kind of object it returns before you use the contents of the object. For example, it may return an `InputStream` object, and in that case, you will need to read data from the input stream. The following are the two versions of the `getContent()` method:

```
public final Object getContent() throws IOException
public final Object getContent(Class[] classes) throws IOException
```

The second version of the `getContent()` method lets you pass an array of class type. It will attempt to convert the content object to one of the classes you pass to it in order. If the content object does not match any of the type, it will return `null`. You will still need to write `if` statements to know what type of object was returned from the `getContent()` method.

```
URL baseURL = new URL ("your url string goes here");
Class[] c = new Class[] { String.class,
                          BufferedReader.class,
                          InputStream.class
                        };
```

```
Object content = baseURL.getContent(c);
if (content == null) {
     // content is not of any of the three kinds
}
else if (content instanceof String) {
     // We got a string
}
else if (content instanceof BufferedReader) {
     // We got a reader
}
else if (content instanceof InputStream) {
     // We got an input stream
}
```

If we read the contents of a URL using the `openStream()` or `getContent()` method, the `URL` class handles many of the complexities of using sockets internally. The downside of this approach is that we do not have any control over the connection settings. We cannot write data to a URL using this approach. Also, we do not have access to the header information for the protocol used in a URL. Don't despair, Java provides another class, `URLConnection`, which lets you do all of these things in a simple and concise manner. `URLConnection` is an `abstract` class and you cannot create its object directly. You need to use the `openConnection()` method of a URL object to get a `URLConnection` object. The `URL` class will handle the creation of a `URLConnection` object, which will be appropriate to handle the data for the protocol used in the URL. The following snippet of code shows how to use a `URLConnection` object to read and write data to a URL.

```
URL url = new URL("your URL string goes here");

// Get a connection object
URLConnection connection = url.openConnection();

// Indicate that you will be writing to the connection
connection.setDoOutput(true);

// Get output/input streams to write/read data
OutputStream ous = connection.getOutputStream();
InputStream ins = connection.getInputStream(); // Caution. Read below
```

The `openConnection()` method of the `URL` class returns a `URLConnection` object, which is not connected to the URL source yet. You must set all connection related parameters to this object before it is connected. For example, if you want to write data to the URL, you must call the `setDoOutput(true)` method on the connection object before it is connected. A `URLConnection` object gets connected when you call its `connect()` method. However, it is connected implicitly, when you call its methods that require a connection. For example, writing data to a URL and reading a URL's data or header fields will connect the `URLConnection` object automatically, if it is not already connected.

Here are few things you must follow if you want to avoid problems when you work with a `URLConnection` to read and write data to a URL.

- When you are only reading data from a URL, you can get the input stream using its `getInputStream()` method. Use the input stream to read data. It will use a GET method for the request to the remote host. That is, if you are passing some parameters to the URL, you must do so by adding the query part to the URL.
- If you are writing as well as reading data from a URL, you must call the `setDoOutput(true)` before you connect. You must finish writing the data to the URL before you start reading the

data. Writing data to a URL will change the request method to `POST`. You cannot even get the input stream before you finish writing data to the URL. In fact, `getInputStream()` method sends a request to the remote host. Your intention is to send the data to the remote host and read the response from the remote host. This one gets as tricky as it could be. Here is a little more explanation, using a snippet of code, assuming that `connection` is a `URLConnection` object.

```
// Wrong – 1. Get input and output streams
// you must get the output stream first
InputStream ins = connection.getInputStream();
OutputStream ous = connection.getOutputStream();


// Wrong – 2. Get output and input streams
// you must get the output stream and finish writing
// before you should get the input stream
OutputStream ous = connection.getOutputStream();
InputStream ins = connection.getInputStream();

// Right. Get output stream and get done with it
// Get the input stream and read data
OutputStream ous = connection.getOutputStream();

// Write logic to write data using ous object here. Make sure
// you are done writing data before you call the
// getInputStream() method as shown below
InputStream ins = connection.getInputStream();

// Write logic to read data
```

- Using the `getInputStream()` method and reading header fields, using any method such as `getHeaderField(String headerName)`, have the same effect. The URL's server supplies both header and content. A `URLConnection` must send the request to get them.

Listing 3-13 has the complete code that writes/reads data to/from echo_params.jsp page as listed in Listing 3-11. This time, we are using the POST method to send data to a URL. Note that the data that we send has been encoded using the `URLEncoder` class. We only needed to encode the value of the name field, which is `"John & Co."`, because the ampersand (&) in the value will conflict with the name-value pair separator in the query string. The program has plenty of comments to warn you of any dangers if you change the sequence of any statements.

The program prints information about all headers that is returned in a `java.util.Map` object. The `URLConnection` class provides you several ways to get the header field's values. For commonly used headers, it provides a direct method. For example, the methods - `getContentLength()`, `getContentType()`, and `getContentEncoding()`, return the value of the header fields that indicate length, type and encoding of the URL's contents, respectively. If you know the header field name or its index, you can use the `getHeaderField(String headerName)` or `getHeaderField(int headerIndex)` method to get its value. The `getHeaderFields()` method returns a `Map` object whose keys represent the header field names and the values represent the header field values. Use caution when reading a header field because it has the same effect on the `URLConnection` object as reading the contents. If you wish to write data to a URL, you must first write the data before you can read the header fields.

Java lets you read the contents of a jar file using a `jar` protocol. Suppose you have a JAR file called *myclasses.jar*, which has a class file whose path is *myfolder/Abc.class*. You can get a `JarURLConnection` from a URL and use its methods to access the JAR file data. Note that you can only read JAR file contents from a URL. You cannot write to a JAR file URL. The following snippet of code shows how to get a `JarURLConnection` object. You will need to use its methods to get the JAR specific data.

```
String str = "jar:http://www.abc.com/myclasses.jar!/myfolder/Abc.class";
URL url = new URL(str);
JarURLConnection connection = (JarURLConnection)url.openConnection();
// Use the connection object to access any jar related data.
```

TIP

We had many words of caution in this section about using a `URLConnection` object. Here is one more. A `URLConnection` object must be used for only one request. It works on the concept of `obtain-use-and-throw`. If you wish to write or read data from a URL multiple times, you must call the URL's `openConnection()` each time separately.

Listing 3-13: A URL reader/writer class that writes/reads data to/from a URL.

```
// URLConnectionReaderWriter.java
package com.jdojo.chapter3;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.net.URL;
import java.net.URLConnection;
import java.net.URLEncoder;
import java.util.Map;

public class URLConnectionReaderWriter {
     public static String getURLContent(String urlStr, String input) {
          BufferedReader br = null;
          BufferedWriter bw = null;

          try {
               URL url = new URL(urlStr);
               URLConnection connection = url.openConnection();

               // Must call setDoOutput(true) to indicate that
               // we will write to the connection. By default,
               // it is false. By default, setDoInput() is set to true.
               connection.setDoOutput(true);

               //Now, connect to the remote object
               connection.connect();

               // Write data to the URL first before reading the response
               OutputStream ous = connection.getOutputStream();
```

```
            bw = new BufferedWriter(new OutputStreamWriter(ous));
            bw.write(input);
            bw.flush();
            bw.close();

            // Must be placed after writing the data. Otherwise,
            // it will result in error, because if write is performed,
            // read must be performed after the write
            printRequestHeaders(connection);

            InputStream ins = connection.getInputStream();

            // Wrap the input stream into a reader
            br = new BufferedReader(new InputStreamReader(ins));

            StringBuffer sb = new StringBuffer();
            String msg = null;
            while ((msg = br.readLine()) != null) {
                sb.append(msg);
                sb.append("\n"); // Append a new line
            }

            return sb.toString();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            if (br != null) {
                try {
                    br.close();
                }
                catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        // If we arrive here it means there was an error
        return null;
    }

    public static void printRequestHeaders(URLConnection connection) {
        Map headers = connection.getHeaderFields();
        System.out.println("Request Headers are:");
        System.out.println(headers);
        System.out.println();
    }

    public static void main(String[] args) {
        String urlStr = "http://www.jdojo.com/docsapp/echo_params.jsp";
        String query = null;
        try {
            // Encode the query. We need to encode only the value of
            // the name parameter. Other names and values are fine
            query = "id=789" + "&" + "name=" +
                    URLEncoder.encode("John & Co.", "utf-8");
```

```
                // Get the content and display it on the console
                String content = getURLContent(urlStr, query);
                System.out.println(content);
            }
            catch (Exception e) {
                e.printStackTrace();
            }
        }
}
```

Output: (The output has been reformatted for readability.)

```
Request Headers are:
{null=[HTTP/1.1 200 OK], Date=[Fri, 19 Dec 2008 02:15:14 GMT], Content-
Length=[402], Set-Cookie=[JSESSIONID=567B1B9F853DD22DD73AB8452E220E0A;
Path=/examples], Content-Type=[text/html;charset=windows-1252],
Server=[Apache-Coyote/1.1]}


<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
         charset=windows-1252"/>
    <title>Echo Request Method and Parameters</title>
  </head>
  <body>
  <h1>URL Connection Test</h1>
  Request Method: POST
<br/><br/>
<u>List of Parameter Names and Values</u><br/>
Name: name, Value: John & Co.
<br/>
Name: id, Value: 789
<br/>
  </body>
</html>
```

Non-Blocking Socket Programming

In the previous sections, we discussed TCP and UDP sockets. The `connect()`, `accept()`, `read()` and `write()` methods of the `Socket` and `ServerSocket` classes block until the operation is complete. For example, a client socket's thread will be blocked, if it has called the `read()` method to read data from a server. Would it not be nice if we could call the `read()` method on a client socket and start doing something else, until data from the server arrives? When data is available from the server, the client socket would be notified, which will read the data at an appropriate time. Another big issue that we face with socket programming is the scalability of a server application. In the previous sections, we had suggested that you would need to create a new thread to handle each client connection or you would have a pool of threads to handle all client connections. Both ways, you will be creating and maintaining a bunch of threads in your program. Would it not be nice if you do not have to deal with threads in a server program to handle multiple clients? Non-blocking socket channels offer all of these nice features. As always, a good feature has a price tag associated with it, so has the non-blocking socket channel? It has a little bit of a

learning curve. We are all used to programming where things happen sequentially. With non-blocking socket channels, you will need to change your mind-set about the way you think about performing things in a program. Changing a mind-set takes some time. Your program will be performing multiple things, which would not be performed sequentially. If you are learning Java for the first time, you can skip this section and revisit it later when you gain some more experience in writing complex Java programs.

Non-blocking socket channel was introduced in Java 1.4. You would not be able to run any of the examples in this section if you are using Java versions prior to Java 1.4

It is assumed that you have a good understanding of socket programming using `ServerSocket` and `Socket` classes. It is further assumed that you have a basic understanding of *New Input/Output* in Java using buffers and channels.

This section uses some classes that are contained in `java.nio`, `java.nio.channels` and `java.nio.charset` packages.

Let us start with comparing the classes that are involved in blocking and non-blocking socket communications. Table 3-6 lists the main classes that are used in blocking and non-blocking socket applications.

Table 3-6: Comparison of classes involved in blocking and non-blocking socket programming

Classes used in Blocking Socket Based Communication	Classes used in Non-Blocking Socket Based Communication
`ServerSocket`	`ServerSocketChannel` The `ServerSocket` class still exists behind the scenes.
`Socket`	`SocketChannel` The `Socket` class still exists behind the scenes.
`InputStream` `OutputStream`	No corresponding classes exist. A `SocketChannel` is used to read/write data
No corresponding class exists.	`Selector`
No corresponding class exists.	`SelectionKey`

We will work with a `ServerSocketChannel` object primarily to accept a new connection request in a server instead of using a `ServerSocket`. The `ServerSocket` has not disappeared. It is still at play behind the scenes. If you need the reference of the `ServerSocket` object being used internally, you can get it by using the `socket()` method of the `ServerSocketChannel` object. You can think of a `ServerSocketChannel` object as a wrapper for a `ServerSocket` object.

We will work with a `SocketChannel` to communication between a client and a server instead of a `Socket`. A `Socket` object is still at play behind the scenes. You can get the reference of the `Socket` object using the `socket()` method of the `SocketChannel` class. You can think of a `SocketChannel` object as a wrapper for a `Socket` object.

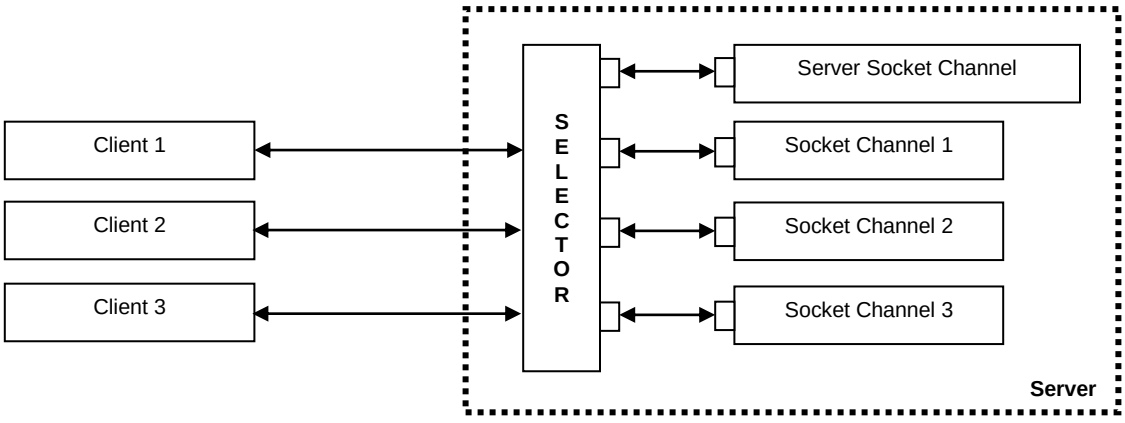
Before we start discussing the mechanism that is used by the non-blocking sockets to give you a more efficient and scalable application interface, it would be beneficial to look at a real world example. Let us discuss the way orders are placed and served in a fast food restaurant. Suppose

the restaurant expects the maximum ten customers and the minimum of zero customers at any time. A customer comes to the restaurant; places his orders; and is served the food. How many servers should that restaurant employ? In the best case, it may employ only one server that can handle receiving orders from all customers and serving their foods. In the worst case, it can have ten servers – one server reserved for one customer. In the latter case, if there are only three customers in the restaurant, seven servers will be idle.

Let us take the middle path in restaurant management. Let us have a few servers in the kitchen to cook and one server at the counter to receive orders. A customer comes; places an order with the server at the counter; the customer gets an order id; the customer leaves the counter; the server at the counter passes on the order to one of the servers in the kitchen and starts taking an order from the next customer. At this point, the customer is free to do something else while his order is being prepared. The server at the counter is dealing with other customers. Servers in the kitchen are busy preparing the food according to the orders placed. No one is waiting for anyone. As soon as the food item in an order is ready, the server at the counter receives it from the server in the kitchen, and calls the order number, so that the customer who placed that order will pick up his food. A customer may get his foods in multiple installments. He can eat the food that he has been served while the remaining items in his order are being prepared in the kitchen. This architecture is the most efficient architecture you can have in a restaurant. It keeps everyone busy most of the time and makes efficient use of the resources. This is the approach that non-blocking socket channels follow. Another approach would be – the customer comes in; places order; waits in queue, until his order is complete and is served; and the next customer places his order and so on. This is the approach blocking sockets follow. If you understand the approach taken by the fast food restaurant for the efficient use of resources, you can understand the non-blocking socket channels easily. We will compare the people used in our restaurant example with objects used in non-blocking sockets in our discussion.

Let us first discuss the situation at the server side. The server side is our restaurant. The person at the counter, who interfaces with all customers, is called a *selector*. A selector is an object of the `Selector` class. Its sole job is to interact with the outside world. It sits between remote clients interacting with the server, and the things inside the server. A remote client never interacts with objects working inside the server, as a customer in the restaurant never interacts directly with servers in the kitchen. Figure 3-7 shows the architecture of non-blocking socket channels communication. It shows where the selector fits into the architecture.

Figure 3-7: Architecture of non-blocking client-server sockets



You cannot create a selector object directly using its constructor. You need to call its `open()` `static` method to get a selector object as shown below.

```
// Get a selector object
Selector selector = Selector.open();
```

A `ServerSocketChannel` is used to listening for a new connection request from clients. Again, you cannot create a new `ServerSocketChannel` object using its constructor. You need to call its `open() static` method as shown below.

```
// Get a server socket channel
ServerSocketChannel ssChannel = ServerSocketChannel.open();
```

By default, a server socket channel or a socket channel is a blocking channel. You need to configure it to make it a non-blocking channel as shown below.

```
// Configure the server socket channel to be non-blocking
ssChannel.configureBlocking(false);
```

Our server socket channel needs to be bound to a local IP address and a local port number, so that a remote client may contact it for new connections. We bind a server socket channel using its `bind()` method. The `bind()` has been added to the `ServerSocketChannel` and the `SocketChannel` in Java 7. Prior to Java 7, you will need to call the `bind()` method on the socket that is associated with the channels.

```
InetAddress hostIPAddress = InetAddress.getByName("localhost");
int port = 19000;

// Prior to Java 7
ssChannel.socket().bind(new InetSocketAddress(hostIPAddress, port));

// Java 7
ssChannel.bind(new InetSocketAddress(hostIPAddress, port));
```

The most important step is taken now. The server socket has to register itself with the selector showing interest in some kind of operation. It is like a server in a restaurant, who wants to make pizza for customers, letting the server at the counter know that he is ready to make Pizza for customers and he needs to be notified when an order for pizza is placed. There are four kinds of operations for which you can register a channel with a selector. They are defined as integer constants in the `SelectionKey` class as listed in Table 3-7. A `ServerSocketChannel` only listens for accepting a new client connection request and therefore it can register for only one operation as shown below.

```
// Register the server socket channel with
// the selector for accept operation
ssChannel.register(selector, SelectionKey.OP_ACCEPT);
```

Table 3-7: List of operations that are recognized by a selector and for which a ServerSocketChannel or a SocketChannel object can register with a selector.

Operation Type	Value	Who Can register for this operation	Description
`Connect`	`SelectionKey.OP_CONNECT`	`SocketChannel` at client	Selector will notify about the connect operation progress.
`Accept`	`SelectionKey.OP_ACCEPT`	`ServerSocketChannel`	Selector will notify

		at server	when a client request for a new connection arrives
Read	SelectionKey.OP_READ	SocketChannel at client and server	Selector will notify when the channel is ready to read some data.
Write	SelectionKey.OP_WRITE	SocketChannel at client and server	Selector will notify when channel is ready to write some data.

The `register()` method of `ServerSocketChannel` returns an object of type `SelectionKey`. You can think of this object as a registration certificate with the selector. You can store this key object in a variable if you need to use it later. Our example ignores it. The selector has a copy of your key (registration details) and it will use it in the future to notify you of any operation for which your channel is ready.

At this point, our selector is ready to intercept an incoming request for a client connection and pass it on to the server socket channel. Suppose a client attempts to connect to the server socket channel at this time. How does interaction between the selector and the server socket channel take place? When the selector detects that there is a registered key with it, which is ready for an operation, it places that key (an object of the `SelectionKey` class) in a separate group, which is called a *ready set*. A `java.util.Set` object represents a ready set. You can determine the number of keys in a ready state by calling the `select()` method of a `Selector` object.

```
int readyCount = selector.select();
```

Once you get at least one ready key in the ready set, you need to get the key and look at the details, You can get all ready keys from the ready set as shown below.

```
// Get the set of ready keys
Set readySet = selector.selectedKeys();
```

Note that you register a key for one or more operations. You need to look at the key details for its readiness for a particular operation. If a key is ready for accepting a new connection request, its `isAccetable()` method will return `true`. If a key is ready for a connection operation, its `isConnectable()` method will return `true`. If a key is ready for read and write operations, its `isReadable()` and `isWritable()` methods will return `true`. You may observe that there is a method to check for the readiness for each operation type. When you are processing a ready set, you will also need to remove the key from the ready set. Here is a typical code that processes a ready set in a server application. An infinite loop is typical on a server application, because you need to keep looking for the next ready set, once you are done with the current ready set.

```
while(true) {
     // Get count of keys in the ready set.
     // If ready key count is greater than zero,
     // process each key in the ready set.
}
```

The following snippet of code shows the typical logic that you can use to process all keys in a ready set.

```
SelectionKey key = null;
Iterator iterator = readySet.iterator();
while (iterator.hasNext()) {
     // Get the next ready selection key object
     key = (SelectionKey)iterator.next();

     // Remove the key from ready set
     iterator.remove();

     // Process the key according to the operation
     if (key.isAcceptable()) {
          // Process new connection
     }

     if (key.isReadable()) {
          // Read from the channel
     }

     if (key.isWritable()) {
          // Write to the channel
     }
}
```

How do you accept a connection request from a remote client on a server socket channel? The logic is similar to accepting a remote connection request using a `ServerSocket` object. A `SelectionKey` object has a reference to the `ServerSocketChannel` that registered it. You can get to the `ServerSocketChannel` object of a `SelectionKey` object using its `channel()` method. You need to call the `accept()` method on the `ServerSocketChannel` object to accept a new connection request. The `accept()` method returns an object of the `SocketChannel` class that is used to communicate (read and write) with a remote client. You need to configure the new `SocketChannel` object to be a non-blocking socket channel. The most important point that you need to understand is that the new `SocketChannel` object must register itself for read, write, or both operations with the selector to start reading/writing data on the connection channel. The following snippet of code shows the logic to accept a remote connection request.

```
ServerSocketChannel ssChannel = (ServerSocketChannel)key.channel();
SocketChannel sChannel = (SocketChannel)ssChannel.accept();
sChannel.configureBlocking(false);

// Register only for read. Our message is small and we
// write it back to the client as soon as we read it
sChannel.register(key.selector(), SelectionKey.OP_READ);
```

If you wish to register the socket channel with a selector for a read and a write, you can do so as shown below.

```
// Register for read and write
sChannel.register(key.selector(),
                  SelectionKey.OP_READ | SelectionKey.OP_WRITE);
```

Once your socket channel is registered with the selector, it will be notified through the selector's ready set, when it receives any data from the remote client, or when you can write data to the remote client on its channel.

If data becomes available on a socket channel, the `key.isReadable()` will return `true` for this socket channel. A typical read operation looks as follows. You must have a basic understanding of Java NIO (New Input/Output) to read data using channels and buffers.

```
SocketChannel sChannel = (SocketChannel) key.channel();
ByteBuffer buffer = ByteBuffer.allocate(1024);
int bytesCount = sChannel.read(buffer);
String msg = "";
if (bytesCount > 0) {
     buffer.flip();
     Charset charset = Charset.forName("UTF-8");
     CharsetDecoder decoder = charset.newDecoder();
     CharBuffer charBuffer = decoder.decode(buffer);
     msg = charBuffer.toString();
     System.out.println("Received Message: " + msg);
}
```

If you can write to a channel, the selector will place the associated key in its ready set, whose `isWritable()` method will return `true`. Again, you need to understand Java NIO to use the `ByteBuffer` object to write data on a channel.

```
SocketChannel sChannel = (SocketChannel)key.channel();
String msg = "message to be sent to remote client goes here";
ByteBuffer buffer = ByteBuffer.wrap(msg.getBytes());
sChannel.write(buffer);
```

What goes on a client side is easy to understand. You start with getting a selector object, and you get a `SocketChannel` object by calling the `SocketChannel.open()` method. At this point, you need to configure the socket channel to be non-blocking, before you connect to the server. Now, you are ready to register your socket channel with the selector. Typically, you register with the selector for connect, read, and write operations. Processing the ready set of the selector is done the same way we processed the ready set of the selector in the server application. The code for reading and writing to the channel will be similar to the server side code. The following snippet of code shows the typical logic used in a client application.

```
InetAddress serverIPAddress = InetAddress.getByName("localhost");
int port = 19000;
InetSocketAddress serverAddress =
                   new InetSocketAddress(serverIPAddress, port);

// Get a selector
Selector selector = Selector.open();

// Create and configure a client socket channel
SocketChannel channel = SocketChannel.open();
channel.configureBlocking(false);

// Connect to the server
channel.connect(serverAddress);

// Register the channel for connect, read and write operations
int operations = SelectionKey.OP_CONNECT | SelectionKey.OP_READ |
                 SelectionKey.OP_WRITE;
channel.register(selector, operations);

// Process the ready set of the selector here...
```

When you get a connect operation on a client side `SocketChannel`, it may mean either a successful or failed connection. You can call the `finishConnect()` method on the `SocketChannel` object to finish the connection process. If the connection has failed, the `finishConnect()` call will throw an exception. Typically, you handle a connect operation as follows.

```
if (key.isConnectable()) {
     try {
          // Call to finishConnect() is in a loop as
          // it is non-blocking for our channel
          while(channel.isConnectionPending()) {
               channel.finishConnect();
          }
     }
     catch (IOException e) {
          // Cancel the channel's registration with the selector
          key.cancel();
          e.printStackTrace();
     }
}
```

It is time to build an echo client application and an echo server application using these channels. Listing 3-14 and Listing 3-15 have the complete code to a non-blocking socket channel for an echo server and an echo client, respectively. You need to run the `NonBlockingEchoServer` class first, and then, one or more instances of the `NonBlockingEchoClient` class. They work similar to our other two echo client-server programs. Note that, this time, you may not see the messages from the server just after you enter a message in the client application. The client application sends a message to the server and it does not wait for the message to be echoed back. Rather, it processes the server message when the socket channel receives the notification from the selector. Therefore, it is possible to get the two messages echoed back from the server at one time. Exception handling has been left out in these examples to keep the code simple and readable.

Listing 3-14: A non-blocking socket channel echo server program

```
// NonBlockingEchoServer.java
package com.jdojo.chapter3;

import java.io.IOException;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import java.util.Iterator;
import java.util.Set;

public class NonBlockingEchoServer {
     public static void main(String[] args) throws Exception {
          InetAddress hostIPAddress = InetAddress.getByName("localhost");
          int port = 19000;
```

```
        // Get a selector
        Selector selector = Selector.open();

        // Get a server socket channel
        ServerSocketChannel ssChannel = ServerSocketChannel.open();

        // Make the server socket channel non-blocking and bind it
        // to an address
        ssChannel.configureBlocking(false);
        ssChannel.socket().bind(
                    new InetSocketAddress(hostIPAddress, port));

        // Register a socket server channel with the
        // selector for accept operation, so that it can be
        // notified when a new connection request arrives
        ssChannel.register(selector, SelectionKey.OP_ACCEPT);

        /* Now we will keep waiting in a loop for any kind of
           request that arrives to the server - connection, read,
           or write request. If a connection request comes in, we will
           accept the request and register a new socket channel with
           the selector for read and write operations. If read or write
           requests come in, we will forward that request to the
           registered channel.
        */
        while (true) {
            if (selector.select() <= 0) {
                continue;
            }
            processReadySet(selector.selectedKeys());
        }
    }

    public static void processReadySet(Set readySet) throws Exception {
        SelectionKey key = null;
        Iterator iterator = null;
        iterator = readySet.iterator();
        while (iterator.hasNext()) {
            // Get the next ready selection key object
            key = (SelectionKey) iterator.next();

            // Remove the key from the ready key set
            iterator.remove();

            // Process the key according to the operation
            // it is ready for
            if (key.isAcceptable()) {
                processAccept(key);
            }

            if (key.isReadable()) {
                String msg = processRead(key);
                if (msg.length() > 0) {
                    echoMsg(key, msg);
                }
            }
        }
```

```
    }

    public static void processAccept(SelectionKey key)
                                                    throws IOException {
        // This method call indicates that we got a new connection
        // request. Accept the connection request and register
        // the new socket channel with the selector, so that client can
        // communicate on a new channel
        ServerSocketChannel ssChannel =
                                (ServerSocketChannel)key.channel();
        SocketChannel sChannel = (SocketChannel) ssChannel.accept();
        sChannel.configureBlocking(false);

        // Register only for read. Our message is small and we
        // write it back to the client as soon as we read it
        sChannel.register(key.selector(), SelectionKey.OP_READ);
    }

    public static String processRead(SelectionKey key)
                                                        throws Exception {
        SocketChannel sChannel = (SocketChannel) key.channel();
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        int bytesCount = sChannel.read(buffer);
        String msg = "";
        if (bytesCount > 0) {
            buffer.flip();
            Charset charset = Charset.forName("UTF-8");
            CharsetDecoder decoder = charset.newDecoder();
            CharBuffer charBuffer = decoder.decode(buffer);
            msg = charBuffer.toString();
            System.out.println("Received Message: " + msg);
        }
        return msg;
    }

    public static void echoMsg(SelectionKey key, String msg)
                                                    throws IOException {
        SocketChannel sChannel = (SocketChannel) key.channel();
        ByteBuffer buffer = ByteBuffer.wrap(msg.getBytes());
        sChannel.write(buffer);
    }
}
```

Listing 3-15: Non-blocking socket channel echo client program

```
// NonBlockingEchoClient.java
package com.jdojo.chapter3;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
```

```
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import java.util.Iterator;
import java.util.Set;

public class NonBlockingEchoClient {
     private static BufferedReader userInputReader = null;

     public static void main(String[] args) throws Exception {
          InetAddress serverIPAddress =
                              InetAddress.getByName("localhost");
          int port = 19000;
          InetSocketAddress serverAddress =
                    new InetSocketAddress(serverIPAddress, port);

          // Get a selector
          Selector selector = Selector.open();

          // Create and configure a client socket channel
          SocketChannel channel = SocketChannel.open();
          channel.configureBlocking(false);
          channel.connect(serverAddress);

          // Register the channel for connect, read and write operations
          int operations = SelectionKey.OP_CONNECT |
                           SelectionKey.OP_READ |
                           SelectionKey.OP_WRITE;
          channel.register(selector, operations);

          userInputReader = new BufferedReader(
                                   new InputStreamReader(System.in));
          while (true) {
               if (selector.select() > 0) {
                    boolean doneStatus =
                         processReadySet(selector.selectedKeys());
                    if (doneStatus) {
                         break;
                    }
               }
          }
          channel.close();
     }

     public static boolean processReadySet(Set readySet)
                                                  throws Exception {
          SelectionKey key = null;
          Iterator iterator = null;
          iterator = readySet.iterator();
          while (iterator.hasNext()) {
               // Get the next ready selection key object
               key = (SelectionKey) iterator.next();

               // Remove the key from the ready key set
               iterator.remove();
```

```
            if (key.isConnectable()) {
                boolean connected = processConnect(key);
                if (!connected) {
                    return true; // Exit
                }
            }

            if (key.isReadable()) {
                String msg = processRead(key);
                System.out.println("[Server]: " + msg);
            }

            if (key.isWritable()) {
                String msg = getUserInput();
                if (msg.equalsIgnoreCase("bye")) {
                    return true; // Exit
                }
                processWrite(key, msg);
            }

        }

        return false; // Not done yet
    }

    public static boolean processConnect(SelectionKey key) {
        SocketChannel channel = (SocketChannel) key.channel();

        try {
            // Call the finishConnect() in a loop as
            // it is non-blocking for our channel
            while (channel.isConnectionPending()) {
                channel.finishConnect();
            }
        }
        catch (IOException e) {
            // Cancel the channel's registration with the selector
            key.cancel();
            e.printStackTrace();
            return false;
        }
        return true;
    }

    public static String processRead(SelectionKey key)
                                                    throws Exception {
        SocketChannel sChannel = (SocketChannel) key.channel();
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        sChannel.read(buffer);
        buffer.flip();
        Charset charset = Charset.forName("UTF-8");
        CharsetDecoder decoder = charset.newDecoder();
        CharBuffer charBuffer = decoder.decode(buffer);
        String msg = charBuffer.toString();
        return msg;
    }
```

```
    public static void processWrite(SelectionKey key, String msg)
                                                    throws IOException {
        SocketChannel sChannel = (SocketChannel) key.channel();
        ByteBuffer buffer = ByteBuffer.wrap(msg.getBytes());
        sChannel.write(buffer);
    }

    public static String getUserInput() throws IOException {
        String promptMsg = "Please enter a message(Bye to quit): ";
        System.out.print(promptMsg);
        String userMsg = userInputReader.readLine();
        return userMsg;
    }
}
```

Socket Security Permissions

You can control the access for a Java program to use sockets using an instance of the `java.net.SocketPermission` class. The generic format used to grant a socket permission in a Java policy file is as follows.

```
grant {
    permission java.net.SocketPermission "target", "actions";
};
```

The target is of the form: `<host name>:<port range>`. The possible values of actions are `accept`, `connect`, `listen`, and `resolve`.

The `listen` action is meaningful only when “localhost” is used as the host name. The `resolve` action refers to DNS lookup and it is implied if any of the other three actions is present.

A host name could be either a DNS name or an IP address. You can use an asterisk (*) as a wildcard character in the DNS host name. If an asterisk is used, it must be used as the leftmost character in the DNS name. If the host name consists only of an asterisk, it refers to any host. The “localhost” for the host name refers to the local machine. You can indicate the port range for the host name in different formats as described below. Here N1 and N2 indicate port numbers (0 to 65535) and it is assumed that N1 is less than N2. Table 3-8 lists the format used for indicating the port range.

Table 3-8: The <port range> format for java.net.SocketPermission security settings

Port Range Value	Description
`N1`	Only one port number - N1
`N1-N2`	Port numbers from N1 to N2
`N1-`	Port numbers from N1 and greater
`-N1`	Port numbers from N1 and less

The following are examples of using a `java.net.SocketPermission` in a Java policy file.

```
// Grant to all codebase
grant {
 // Permission to connect with 192.168.10.123 at port 5000
 permission java.net.SocketPermission "192.168.10.123:5000", "connect";

 // Connect permission to any host at port 80
 permission java.net.SocketPermission "*:80", "connect";

 // All socket permissions to any application on port 1024 or greater
 // on the local machine. The entry below is in one line
 permission java.net.SocketPermission "localhost:1024-", "listen, accept,
connect";
};
```

Asynchronous Socket Channels

Java 7 added support for asynchronous socket operations such as connect, read, and write. The asynchronous socket operations are performed using the following two socket channel classes.

- `java.nio.channels.AsynchronousServerSocketChannel`
- `java.nio.channels.AsynchronousSocketChannel`

An instance of the `AsynchronousServerSocketChannel` class serves as a server socket that listens for new incoming client connections. Once it accepts a new client connection, the interaction between the client and the server is handled by an instance of the `AsynchronousSocketChannel` class at both ends. Asynchronous socket channels are set up very similar to the synchronous sockets. The main difference between the two setups is that the request for an asynchronous socket operation returns immediately and the requestor is notified when the operation is completed, whereas in a synchronous socket operation the request for a socket operation blocks until it is complete. Because of the asynchronous nature of the operations with the asynchronous socket channels, the code to handle the completion or failure of a socket operation gets a little complex.

In an asynchronous socket channel, you request an operation using one of the methods of the above-mentioned asynchronous socket channel classes. The method returns immediately. You receive a notification about the completion or failure of the operation later. The methods that allow you to request asynchronous operations are overloaded. One version returns a `Future` object that lets you check the status of the requested operation. Please refer to the chapter on *Threads* (Vol. 2) for details on using a `Future` object. Another version of those methods lets you pass a `CompletionHandler`. When the requested operation completes successfully, the `completed()` method of the `CompletionHandler` is called. When the requested operation fails, the `failed()` method of the `CompletionHandler` is called. The following snippet of code demonstrates both approaches of handling the completion/failure of a requested asynchronous socket operation. It shows how a server socket channel accepts a client connection asynchronously.

Using a Future Object

```
// Get a server socket channel instance
AsynchronousServerSocketChannel server = get a server instance...;

// Bind the socket to a host and a port
server,bind(your_host, your_port);
```

```
// Start accepting a new client connection. Note that the accept()
// method returns immediately by returning a Future object
Future<AsynchronousSocketChannel> result = server.accept();

// Wait for the new client connection by calling the get() method
// of the Future object. Alternatively, you can poll the Future object
// periodically using its isDone() method
AsynchronousSocketChannel newClient = result.get();

/* Handle the newClient here and call the server.accept() again to
   Accept another client connection
*/
```

Using a CompletionHandler Object

```
// Get a server socket channel instance
AsynchronousServerSocketChannel server = get a server instance...;

// Bind the socket to a host and a port
server,bind(your_host, your_port);

// Start accepting a new client connection. Note that the accept()
// method returns immediately. The completed() or failed() method
// of the ConnectionHandler will be called upon completion or failure
// of the requested operation
YourAnyClass attach = ...; // Get an attachment
server.accept(attach, new ConnectionHandler());
```

The above version of the `accept()` method accepts an object of any class as an attachment. It could be a `null` reference. The attachment is passed to the `completed()` and `failed()` methods of the completion handler, which is an object of `ConnectionHandler` in this case. The `ConnectionHandler` class may look as follows.

```
private static class ConnectionHandler
implements CompletionHandler<AsynchronousSocketChannel, YourAnyClass> {
     @Override
     public void completed(AsynchronousSocketChannel client,
                           YourAnyClass attach) {
          // Handle the new client connection here and again start
          // accepting a new client connection
     }

     @Override
     public void failed(Throwable e, YourAnyClass attach) {
          // Handle the failure here
     }
}
```

In this section, we will cover the following three steps in detail. During our discussion of these steps, we will build an application that consists of an echo server and a client. Clients will send messages to the server asynchronously and the server will echo back the message to the client asynchronously. It is assumed that you have familiarity working with buffers and channels. Please refer to the chapter on *New Input/Output (NIO)* (Vol. 2) for more details on using buffers and channels.

- Setting up an asynchronous server socket channel
- Setting up an asynchronous client socket channel
- Putting the asynchronous server and client socket channels in action

Setting up an Asynchronous Server Socket Channel

An instance of the `AsynchronousServerSocketChannel` class is used as an asynchronous server socket channel to listen to the new incoming client connections. Once a connection to a client is established, an instance of the `AsynchronousSocketChannel` class is used to communicate with the client. The `static open()` method of the `AsynchronousServerSocketChannel` class returns an object of the `AsynchronousServerSocketChannel` class, which is not yet bound.

```
// Create an asynchronous server socket channel object
AsynchronousServerSocketChannel server =
                              AsynchronousServerSocketChannel.open();

// Bind the server to the localhost and the port 8989
String host = "localhost";
int port = 8989;
InetSocketAddress sAddr = new InetSocketAddress(host, port);
server.bind(sAddr);
```

At this point, our server socket channel can be used to accept a new client connection by calling its `accept()` method as follows. The code uses two classes, `Attachment` and `ConnectionHandler`, which are described later.

```
// Prepare the attachment
Attachment attach = new Attachment();
attach.server = server;

// Accept new connections
server.accept(attach, new ConnectionHandler());
```

Typically, a server application runs indefinitely. You can make the server application run forever by waiting on the main thread in the `main()` method as follows.

```
try {
     // Wait indefinitely until someone interrups the main thread
     Thread.currentThread().join();
}
catch (InterruptedException e) {
     e.printStackTrace();
}
```

We will use the completion handler mechanism to handle the completion/failure notification for the server socket channel. An object of the following `Attachment` class will be used to serve as an attachment to the completion handler. An attachment object is used to pass the context for the server socket that may be used inside the `completed()` and `failed()` methods of the completion handler.

```
class Attachment {
     AsynchronousServerSocketChannel server;
```

```
     AsynchronousSocketChannel client;
     ByteBuffer buffer;
     SocketAddress clientAddr;
     boolean isRead;
}
```

We need a `CompletionHandler` implementation to handle the completion of an `accept()` call. Let us call our class as `ConnectionHandler` as shown below.

```
private static class ConnectionHandler
implements CompletionHandler<AsynchronousSocketChannel, Attachment> {
     @Override
     public void completed(AsynchronousSocketChannel client,
                           Attachment attach) {
          try {
               // Get the client address
               SocketAddress clientAddr = client.getRemoteAddress();

               System.out.format("Accepted a connection from %s%n",
                                 clientAddr);

               // Accept another connection
               attach.server.accept(attach, this);

               // Handle the client connection by invoking an asyn read
               Attachment newAttach = new Attachment();
               newAttach.server = attach.server;
               newAttach.client = client;
               newAttach.buffer = ByteBuffer.allocate(2048);
               newAttach.isRead = true;
               newAttach.clientAddr = clientAddr;

               // Create a new completion handler for reading to and
               // writing from the new client
               ReadWriteHandler readWriteHandler = new ReadWriteHandler();

               // Read from the client
               client.read(newAttach.buffer, newAttach, readWriteHandler);
          }
          catch (IOException e) {
               e.printStackTrace();
          }

     }

     @Override
     public void failed(Throwable e, Attachment attach) {
          System.out.println("Failed to accept a connection.");
          e.printStackTrace();
     }
}
```

The `ConnectionHandler` class is simple. In its `failed()` method, it prints the exception stack trace. In its `completed()` method, it prints a message that a new client connection has been established and starts listening for another new client connection by calling the `accept()` method on the server socket again. Note the reuse of the attachment in another `accept()` method call

inside the `completed()` method. It uses the same `CompletionHandler` object again. Note that the `attach.server.accept(attach, this)` method call uses the keyword `this` to refer to the same instance of the completion handler. At the end, it prepares a new instance of the `Attachment` class, which wraps details of handling (reading and writing) the new client connection, and calls the `read()` method on the client socket to read from the client. Note that the `read()` method uses another completion handler, which is an instance of the `ReadWriteHandler` class. The code for the `ReadWriteHandler` is as follows.

```
private static class ReadWriteHandler
          implements CompletionHandler<Integer, Attachment> {
     @Override
     public void completed(Integer result, Attachment attach) {
          if (result == -1) {
               try {
                    attach.client.close();
                    System.out.format("Stopped listening to" +
                                      " the client %s%n",
                                       attach.clientAddr);
                    }
                    catch (IOException ex) {
                         ex.printStackTrace();
                    }
                    return;
               }

          if (attach.isRead) {
               /* A read to the client was completed */

               // Get the buffer ready to read from it
               attach.buffer.flip();

               int limits = attach.buffer.limit();
               byte bytes[] = new byte[limits];
               attach.buffer.get(bytes, 0, limits);
               Charset cs = Charset.forName("UTF-8");
               String msg = new String(bytes, cs);

               // Print the message from the client
               System.out.format("Client at %s says: %s%n",
                                 attach.clientAddr, msg);

               // Let us echo back the same message to the client
               attach.isRead = false; // It is a write

               // Prepare the buffer to be read again
               attach.buffer.rewind();

               // Write to the client again
               attach.client.write(attach.buffer, attach, this);
          }
          else {
               // A write to the client was completed
               // Perform another read from the client
               attach.isRead = true;

               // Prepare the buffer to be filled in
```

```
                attach.buffer.clear();

                // Perform a read from the client
                attach.client.read(attach.buffer, attach, this);
            }
        }

        @Override
        public void failed(Throwable e, Attachment attach) {
            e.printStackTrace();
        }
}
```

The first argument, `result`, of the `completed()` method is the number of bytes that is read from or written to the client. Its value of -1 indicates the end-of-stream, and in that case, the client socket is closed. If a read operation was completed, it displays the read text on the standard output and writes back the same text to the client. If a write operation to a client was completed, it performs a read on the same client.

Listing 3-16 has the complete code for our asynchronous server socket channel. It uses three inner classes – one for the attachment, one for the connection completion handler, and one for the read/write completion handler. The `AsyncEchoServerSocket` class can be run now. However, it will not do any work as it needs a client to connect to it to echo back messages that are sent from the client. We will develop our asynchronous client socket channel in the next section, and then, in the subsequent section, we will test both server and client socket channels together.

Listing 3-16: A server application that uses asynchronous server socket channel

```
// AsyncEchoServerSocket.java
package com.jdojo.chapter3;

import java.io.IOException;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.charset.Charset;
import java.net.InetSocketAddress;
import java.nio.channels.CompletionHandler;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.AsynchronousServerSocketChannel;

public class AsyncEchoServerSocket {
    private static class Attachment {
        AsynchronousServerSocketChannel server;
        AsynchronousSocketChannel client;
        ByteBuffer buffer;
        SocketAddress clientAddr;
        boolean isRead;
    }

    private static class ConnectionHandler
          implements CompletionHandler<AsynchronousSocketChannel,
                                       Attachment> {
        @Override
        public void completed(AsynchronousSocketChannel client,
                              Attachment attach) {
            try {
```

```
                // Get the client address
                SocketAddress clientAddr = client.getRemoteAddress();
                System.out.format("Accepted a connection from %s%n",
                                  clientAddr);

                // Accept another connection
                attach.server.accept(attach, this);

                // Handle the client connection by using an asyn read
                ReadWriteHandler rwHandler = new ReadWriteHandler();
                Attachment newAttach = new Attachment();
                newAttach.server = attach.server;
                newAttach.client = client;
                newAttach.buffer = ByteBuffer.allocate(2048);
                newAttach.isRead = true;
                newAttach.clientAddr = clientAddr;
                client.read(newAttach.buffer, newAttach, rwHandler);
            }
            catch (IOException e) {
                e.printStackTrace();
            }

        }

        @Override
        public void failed(Throwable e, Attachment attach) {
            System.out.println("Failed to accept a connection.");
            e.printStackTrace();
        }
    }

    private static class ReadWriteHandler
        implements CompletionHandler<Integer, Attachment> {
        @Override
        public void completed(Integer result, Attachment attach) {
            if (result == -1) {
                try {
                    attach.client.close();
                    System.out.format(
                         "Stopped listening to the client %s%n",
                          attach.clientAddr);
                }
                catch (IOException ex) {
                    ex.printStackTrace();
                }
                return;
            }

            if (attach.isRead) {
                /* A read to the client was completed */

                // Get the buffer ready to read from it
                attach.buffer.flip();

                int limits = attach.buffer.limit();
                byte bytes[] = new byte[limits];
                attach.buffer.get(bytes, 0, limits);
```

```
                Charset cs = Charset.forName("UTF-8");
                String msg = new String(bytes, cs);

                // Print the message from the client
                System.out.format("Client at %s says: %s%n",
                                  attach.clientAddr, msg);

                // Let us echo back the same message to the client
                attach.isRead = false; // It is a write

                // Prepare the buffer to be read again
                attach.buffer.rewind();

                // Write to the client
                attach.client.write(attach.buffer, attach, this);
            }
            else {
                // A write to the client was completed
                // Perform another read from the client
                attach.isRead = true;

                // Prepare the buffer to be filled in
                attach.buffer.clear();

                // Perform a read from the client
                attach.client.read(attach.buffer, attach, this);
            }
        }

        @Override
        public void failed(Throwable e, Attachment attach) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        try (AsynchronousServerSocketChannel server =
                      AsynchronousServerSocketChannel.open()) {

            // Bind the server to the localhost and the port 8989
            String host = "localhost";
            int port = 8989;
            InetSocketAddress sAddr =
                          new InetSocketAddress(host, port);
            server.bind(sAddr);

            // Display a message that server is ready
            System.out.format("Server is listening at %s%n", sAddr);

            // Prepare the attachment
            Attachment attach = new Attachment();
            attach.server = server;

            // Accept new connections
            server.accept(attach, new ConnectionHandler());

            try {
```

```
                    // Wait until the main thread is interrupted
                    Thread.currentThread().join();
               }
               catch (InterruptedException e) {
                    e.printStackTrace();
               }
          }
          catch (IOException e) {
               e.printStackTrace();
          }
     }
}
```

Setting up an Asynchronous Client Socket Channel

An instance of the `AsynchronousSocketChannel` class is used as an asynchronous client socket channel in a client application. The `static open()` method of the `AsynchronousSocketChannel` class returns an open channel of the `AsynchronousSocketChannel` type, which is not yet connected to a server socket channel. The channel's `connect()` method is used to connect to a server socket channel. The following snippet of code shows how to create an asynchronous client socket channel and connect it to a server socket channel. It uses a `Future` object to handle the completion of the connection to the server.

```
// Create an asynchronous socket channel
AsynchronousSocketChannel channel = AsynchronousSocketChannel.open();

// Connect the channel to the server
String serverName = "localhost";
int serverPort = 8989;
SocketAddress serverAddr = new InetSocketAddress(serverName, serverPort);

Future<Void> result = channel.connect(serverAddr);
System.out.println("Connecting to the server...");

// Wait for the connection to complete
result.get();

// Connection to the server is complete now
System.out.println("Connected to the server...");
```

Once the client socket channel is connected to a server, you can start reading from the server and writing to the server using the channel's `read()` and `write()` methods asynchronously. Both methods let you handle the completion of the operation using a `Future` object or a `CompletionHandler` object. We will use an `Attachment` class as shown below to pass the context to the completion handler.

```
class Attachment {
     AsynchronousSocketChannel channel;
     ByteBuffer buffer;
     Thread mainThread;
     boolean isRead;
}
```

In the `Attachment` class, the `channel` instance variable holds the reference to the client channel. The `buffer` instance variable holds the reference to the data buffer. We will reuse the same data buffer for reading and writing. The `mainThread` instance variable holds the reference to the main thread of the application. When the client channel is done, we can interrupt the waiting main thread, so that the client application terminates. The `isRead` instance variable indicates if the operation is a read or a write. If it is `true`, it means it is a read operation. Otherwise, it is a write operation.

Listing 3-17 has the complete code for an asynchronous client socket channel. It uses two inner classes - `Attachment` and `ReadWriteHandler`. An instance of the `Attachment` class is used as an attachment to the `read()` and `write()` asynchronous operations. An instance of the `ReadWriteHandler` class is used as a completion handler for the `read()` and `write()` operations. Its `getTextFromUser()` method prompts the user to enter a message on the standard input and returns the user entered message. The `completed()` method of the completion handler checks if it is a read or a write operation. If it is a read operation, it prints the text that was read from the server on the standard output. It prompts the user for another message. If the user enters “Bye”, it terminates the application by interrupting the waiting main thread. Note that the channel is closed automatically, when the program exits the `try` block, because it is opened inside a `try-with-resources` block in the `main()` method.

Listing 3-17: An asynchronous client socket channel

```
// AsyncEchoClientSocket.java
package com.jdojo.chapter3;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.charset.Charset;
import java.util.concurrent.Future;
import java.nio.channels.CompletionHandler;
import java.util.concurrent.ExecutionException;
import java.nio.channels.AsynchronousSocketChannel;

public class AsyncEchoClientSocket {
     private static class Attachment {
          AsynchronousSocketChannel channel;
          ByteBuffer buffer;
          Thread mainThread;
          boolean isRead;
     }

     private static class ReadWriteHandler
          implements CompletionHandler<Integer, Attachment> {

          @Override
          public void completed(Integer result, Attachment attach) {
               if (attach.isRead) {
                    attach.buffer.flip();

                    // Get the text read from the server
                    Charset cs = Charset.forName("UTF-8");
```

```
            int limits = attach.buffer.limit();
            byte bytes[] = new byte[limits];
            attach.buffer.get(bytes, 0, limits);
            String msg = new String(bytes, cs);

            // A read from the server was completed
            System.out.format("Server Responded: %s%n", msg);

            // Prompt the user for another message
            msg = this.getTextFromUser();
            if (msg.equalsIgnoreCase("bye")) {
                 // Interrupt the main thread, so that the program
                 // terminates
                 attach.mainThread.interrupt();
                 return;
            }

            // Prepare buffer to be filled in again
            attach.buffer.clear();
            byte[] data = msg.getBytes(cs);
            attach.buffer.put(data);

            // Prepared buffer to be read
            attach.buffer.flip();

            attach.isRead = false; // It is a write

            // Write to the server
            attach.channel.write(attach.buffer, attach, this);
       }
       else {
            // A write to the server was completed
            // Perform another read from the server
            attach.isRead = true;

            // Prepare the buffer to be filled in
            attach.buffer.clear();

            // Read from the server
            attach.channel.read(attach.buffer, attach, this);
       }
  }

  @Override
  public void failed(Throwable e, Attachment attach) {
       e.printStackTrace();
  }

  private String getTextFromUser() {
       System.out.print("Please enter a message (Bye to quit):");
       String msg = null;

       BufferedReader consoleReader =
            new BufferedReader(new InputStreamReader(System.in));
       try {
            msg = consoleReader.readLine();
       }
```

```
                catch (IOException e) {
                    e.printStackTrace();
                }

                return msg;
            }
        }

        public static void main(String[] args) {
            // Use a try-with-resources to open a channel
            try (AsynchronousSocketChannel channel
                    = AsynchronousSocketChannel.open()) {
                // Connect the client to the server
                String serverName = "localhost";
                int serverPort = 8989;
                SocketAddress serverAddr =
                        new InetSocketAddress(serverName, serverPort);

                Future<Void> result = channel.connect(serverAddr);
                System.out.println("Connecting to the server...");

                // Wait for the connection to complete
                result.get();

                // Connection to the server is complete now
                System.out.println("Connected to the server...");

                // Start reading from and writing to the server
                Attachment attach = new Attachment();
                attach.channel = channel;
                attach.buffer = ByteBuffer.allocate(2048);
                attach.isRead = false;
                attach.mainThread = Thread.currentThread();

                // Place the "Hello" message in the buffer
                Charset cs = Charset.forName("UTF-8");
                String msg = "Hello";
                byte[] data = msg.getBytes(cs);
                attach.buffer.put(data);
                attach.buffer.flip();

                // Write to the server
                ReadWriteHandler readWriteHandler = new ReadWriteHandler();
                channel.write(attach.buffer, attach, readWriteHandler) ;

                // Let this thread wait for ever on its own death
                // until interrupted
                attach.mainThread.join();
            }
            catch (ExecutionException | IOException e) {
                e.printStackTrace();
            }
            catch(InterruptedException e) {
                System.out.println("Disconnected from the server.");
            }
        }
    }
```

Putting the Server and the Client Together

At this point, our asynchronous server and client programs are ready. You need to use the following steps to run the server and the client

Step-1

Run the `AsyncEchoServerSocket` class as listed in Listing 3-16. You should get a message on the standard output as shown below.

```
Server is listening at localhost/127.0.0.1:8989
```

If you get the above message, you need to proceed to the next step. If you do not get the above message, it is most likely that the port 8989 is being used by another process. In such a case, you should get the following error message.

```
java.net.BindException: Address already in use: bind
```

If you get "Address already in use" error message, you need to change the port value in the `AsyncEchoServerSocket` class from 8989 to some other value and retry running the `AsyncEchoServerSocket` class. If you change the port number in the server program, you must also change the port number in the client program to match with the server port number. The server socket channel listens at a port and the client must connect to the same port on which the server is listening.

Step-2

Before proceeding with this step, make sure that you were able to run step-1 successfully. Run the `AsyncEchoClientSocket` class as listed in Listing 3-17. You can run one or more instances of the `AsyncEchoClientSocket` class. You should get the following message on the standard output if the client application was able to connect to the server successfully.

```
Connecting to the server...
Connected to the server...
Server Responded: Hello
Please enter a message (Bye to quit):
```

You might receive the following error message, when you attempt to run the `AsyncEchoClientSocket` class.

```
Connecting to the server...
java.util.concurrent.ExecutionException: java.io.IOException: The remote
system refused the network connection.
```

Typically, this error message indicates one of the following problems:

- The server is not running. If this is the case, make sure that server is running.
- The client is attempting to connect to the server on a different host and port than the host and the port on which the server is listening. If this is the case, make sure that the server and the client are using the same host names (or IP addresses) and the port numbers.

If you were able to run the server and the client programs successfully, you would get outputs on the server and the client that are similar to the ones shown below. You will need to stop the server program manually, e.g., by pressing `Ctrl + C` keys on a DOS prompt.

`Output:` (Client 1)

```
Connecting to the server...
Connected to the server...
Server Responded: Hello
Please enter a message (Bye to quit):How is life at server side?
Server Responded: How is life at server side?
Please enter a message (Bye to quit):Bye
Disconnected from the server.
```

`Output:` (Client 2)

```
Connecting to the server...
Connected to the server...
Server Responded: Hello
Please enter a message (Bye to quit):Async socket channels are cool!
Server Responded: Async socket channels are cool!
Please enter a message (Bye to quit):Bye
Disconnected from the server.
```

`Output:` (Server)

```
Server is listening at localhost/127.0.0.1:8989
Accepted a connection from /127.0.0.1:1436
Client at /127.0.0.1:1436 says: Hello
Accepted a connection from /127.0.0.1:1437
Client at /127.0.0.1:1437 says: Hello
Client at /127.0.0.1:1436 says: How is life at server side?
Client at /127.0.0.1:1437 says: Async socket channels are cool!
Stopped listening to the client /127.0.0.1:1437
Stopped listening to the client /127.0.0.1:1436
```

Datagram-Oriented Socket Channels

Java 1.4 added support for a datagram channel, which is a datagram-oriented selectable channel. An instance of the `java.nio.channels.DatagramChannel` class represents a datagram channel. By default, it is blocking. You can configure it as non-blocking by using its `configureBlocking(false)` method.

Java 7 added multicasting capability to the datagram channel. To create a `DatagramChannel`, you need to invoke one of its `open() static` methods. If you want to use it for IP multicasting, you need to specify the address type (or protocol family) of the multicast group as an argument to its `open()` method. The `open()` method creates a `DatagramChannel` object, which is not connected. If you want your datagram channel to send and receive datagrams only to a specific remote host, you need to use its `connect()` method to connect the channel to that specific host. A datagram channel, which is not connected, may send datagrams to and receive datagrams from any remote host. The following steps are typically needed to send/receive datagrams using a datagram channel.

Step-1

Create a datagram channel using the `open()` method of the `DatagramChannel` class. The following snippet of code shows three different ways to create a datagram channel.

```
// Create a new datagram channel to send/receive datagram
DatagramChannel channel = DatagramChannel.open();

// Create a datagram channel to receive datagrams from a multicast group
// that uses IPv4 address type
DatagramChannel ipv4MulticastChannel =
                    DatagramChannel.open(StandardProtocolFamily.INET);

// Create a datagram channel to receive datagrams from a multicast group
// that uses IPv6 address type
DatagramChannel iPv6MulticastChannel =
                    DatagramChannel.open(StandardProtocolFamily.INET6);
```

Step-2

Set the channel options using the `setOption()` method of the `DatagramChannel` class. Some options must be set before binding the channel to a specific address, whereas some can be set after binding. The `setOption()` method was added to the `DatagramChannel` class in Java 7. If you are using a prior Java version, you will need to use the `socket()` method to get the `DatagramSocket` reference and use one of the methods of the `DatagramSocket` class to set the channel options. The following snippet of code shows how to set the channel options. Table 3-9 contains the list of socket options and their descriptions.

```
/* To bind multiple sockets to the same socket address, you need to
   set the SO_REUSEADDR option for the socket
*/
// In Java 7
channel.setOption(StandardSocketOptions.SO_REUSEADDR, true)

// In Java 1.4
DatagramSocket socket = channel.socket();
socket.setReuseAddress(true);
```

Step-3

Bind the datagram channel to a specific local address and port using the `bind()` method of the `DatagramChannel` class. If you use `null` as the bind address, this method will bind the socket to an available address automatically. The `bind()` method was added to the `DatagramChannel` class in Java 7. If you are using a prior Java version, you can bind a datagram channel using its underlying socket. The following snippet of code shows how to bind a datagram channel.

```
/* In Java 7 *
// Bind the channel to any available address automatically
channel.bind(null);

// Bind the channel to "localhost" and port 8989
InetSocketAddress sAddr = new InetSocketAddress("localhost", 8989);
channel.bind(sAddr);

/* In Java 1.4 */
```

```
// Get the socket reference
DatagramSocket socket = channel.socket();

// Bind the channel to any available address automatically
socket.bind(null);

// Bind the channel to "localhost" and port 8989
InetSocketAddress sAddr = new InetSocketAddress("localhost", 8989);
socket.bind(sAddr);
```

Step-4

To send a datagram to a remote host, use the `send()` method of the `DatagramChannel` class. The method accepts a `ByteBuffer` and a remote `SocketAddress`. If you call the `send()` method on an unbound datagram channel, the `send()` method binds the channel automatically to an available address.

```
// Prepare a message to send
String msg = "Hello";
ByteBuffer buffer = ByteBuffer.wrap(msg.getBytes());

// Pack the remote address and port into an object
InetSocketAddress serverAddress =
                        new InetSocketAddress("localhost", 8989);

// Send the message to the remote host
channel.send(buffer, serverAddress);
```

The `receive()` method of the `DatagramChannel` class lets a datagram channel receive a datagram from a remote host. This method requires you to provide a `ByteBuffer` to receive the data. The received data is copied to the specified `ByteBuffer` at its current position. If the `ByteBuffer` has less space available than the received data, the extra data is discarded silently. The `receive()` method returns the address of the remote host. If the datagram channel is in a non-blocking mode, the `receive()` method returns immediately by retuning `null`. Otherwise, it waits until it receives a datagram.

```
// Prepare a ByteBufer to receive data
ByteBuffer buffer = ByteBuffer.allocate(1024);

// Wait to receive data from a remote host
SocketAddress remoteAddress = channel.receive(buffer);
```

Step-5

Finally, close the datagram channel using its `close()` method.

```
channel.close();
```

Table 3-9: List of standard socket options defined as constants in the java.net.StandardSocketOptions class

Socket Option Name	Description
`SO_SNDBUF`	The size of the socket send buffer in bytes. Its value is of `Integer` type.
`SO_RCVBUF`	The size of the socket receive buffer in bytes. Its value is of `Integer` type.

SO_REUSEADDR	For datagram sockets, it allows multiple programs to bind to the same address. Its value is of `Boolean` type. This option should be enabled for IP multicasting using the datagram channels.
SO_BROADCAST	Allows transmission of broadcast datagrams. Its value is of type `Boolean`.
IP_TOS	The Type of Service (ToS) octet in the Internet Protocol (IP) header Its value is of `Integer` type.
IP_MULTICAST_IF	The network interface for Internet Protocol (IP) multicast datagrams. Its value is a reference of `java.net.NetworkInterface` type.
IP_MULTICAST_TTL	The time-to-live for Internet Protocol (IP) multicast datagrams. Its value is of type `Integer` in the range of 0 to 255.
IP_MULTICAST_LOOP	Loopback for Internet Protocol (IP) multicast datagrams. Its value if of type `Boolean`.

TIP

If a datagram channel is connected to a remote host, it can use the `read()` and `write()` methods of the `DatagramChannel` class to read from and write to the remote host.

Listing 3-18 has a program that acts as an echo server. Listing 3-19 has a program that acts as a client. The echo server waits for a message from a remote client. It echoes the message that it receives from the remote client. You need to start the echo server program before starting the client program. You can run multiple client programs simultaneously.

Listing 3-18: An echo server based on the datagram channel

```
// DGCEchoServer.java
package com.jdojo.chapter3;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;

public class DGCEchoServer {
     public static void main(String[] args) {
          DatagramChannel server = null;

          try {
               // Create a datagram channel and bind it to localhost at
               // port 8989
               server = DatagramChannel.open();
               InetSocketAddress sAddr =
                              new InetSocketAddress("localhost", 8989);
               server.bind(sAddr);

               ByteBuffer buffer = ByteBuffer.allocate(1024);

               // Wait in an infinite loop for a client to send data
               while (true) {
                    System.out.println("Waiting for a message from" +
                                       " a remote host at " + sAddr);
```

```
                    // Wait for a client to send a message
                    SocketAddress remoteAddr = server.receive(buffer);

                    // Prepare the buffer to read the message
                    buffer.flip();

                    // Convert the buffer data into a String
                    int limits = buffer.limit();
                    byte bytes[] = new byte[limits];
                    buffer.get(bytes, 0, limits);
                    String msg = new String(bytes);

                    System.out.println("Client at " + remoteAddr +
                                       " says: " + msg);

                    // Reuse the buffer to echo the message to the client
                    buffer.rewind();

                    // Send the message back to the client
                    server.send(buffer, remoteAddr);

                    // Prepare the buffer to receive the next message
                    buffer.clear();
               }
          }
          catch (IOException e) {
               e.printStackTrace();
          }
          finally {
               // Close the channel
               if (server != null) {
                    try {
                         server.close();
                    }
                    catch (IOException e) {
                         e.printStackTrace();
                    }
               }
          }
     }
}
```

Listing 3-19: A client program based on the datagram channel

```
// DGCEchoClient.java
package com.jdojo.chapter3;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;

public class DGCEchoClient {
     public static void main(String[] args) {
          DatagramChannel client = null;
          try {
```

```
            // Create a new datagram channel
            client = DatagramChannel.open();

            // Bind the client to any available local address and port
            client.bind(null);

            // Prepare a message for the server
            String msg = "Hello";
            ByteBuffer buffer = ByteBuffer.wrap(msg.getBytes());
            InetSocketAddress serverAddress =
                        new InetSocketAddress("localhost", 8989);

            // Send the message to the server
            client.send(buffer, serverAddress);

            // Reuse the buffer to receive a response from the server
            buffer.clear();

            // Wait for the server to respond
            client.receive(buffer);

            // Prepare the buffer to read the message
            buffer.flip();

            // Convert the buffer into a string
            int limits = buffer.limit();
            byte bytes[] = new byte[limits];
            buffer.get(bytes, 0, limits);
            String response = new String(bytes);

            // Print the server message on the standard output
            System.out.println("Server responded: " + response);
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        finally {
            // Close the channel
            if (client != null) {
                try {
                    client.close();
                }
                catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Multicasting Using Datagram Channels

Java 7 added support for IP multicasting to a datagram channel. A datagram channel that is interested in receiving multicast datagrams joins a multicast group. The datagrams that are sent to

a multicast group are delivered to all its members. The following steps are typically needed to set up a client application that is interested in receiving a multicast datagram.

Step-1

Create a datagram channel to use a specific multicast address type as follows. In your application, you will be using IPv4 or IPv6 and not both.

```
// Need to use INET protocol family for an IPv4 addressing scheme
DatagramChannel client =
              DatagramChannel.open(StandardProtocolFamily.INET);

// Need to use INET6 protocol family for an IPv6 addressing scheme
DatagramChannel client =
              DatagramChannel.open(StandardProtocolFamily.INET6);
```

Step-2

Set the options for the client channel.

```
// Let other sockets reuse the same address
client.setOption(StandardSocketOptions.SO_REUSEADDR, true);
```

Step-3

Bind the client channel to a local address and a port.

```
int MULTICAST_PORT = 8989;
client.bind(new InetSocketAddress(MULTICAST_PORT));
```

Step-4

Set the socket option `IP_MULTICAST_IF` that specifies the network interface on which the client channel will join the multicast group.

```
// Get the reference of a network interface named "eth0"
NetworkInterface interf = NetworkInterface.getByName("eth0");

// Set the IP_MULTICAST_IF option
client.setOption(StandardSocketOptions.IP_MULTICAST_IF, interf);
```

You can print the name and description of all network interfaces that are available on your machine using the following snippet of code.

```
import java.net.NetworkInterface;
import java.net.SocketException;
import java.util.Enumeration;

// Other logic goes here...

try {
     Enumeration<NetworkInterface> e =
                              NetworkInterface.getNetworkInterfaces();
     while(e.hasMoreElements()) {
          NetworkInterface nif = e.nextElement();
```

```
            System.out.println("Name: " + nif.getName() +
                               ", Description:" + nif.getDisplayName());
        }
    }
    catch (SocketException ex) {
        ex.printStackTrace();
    }
```

Output: (You may get a different output.)

```
Name: lo, Description:MS TCP Loopback interface
Name: eth0, Description:Intel(R) 82567LM-3 Gigabit Network Connection -
Teefer2 Miniport
```

Step-5

Now, it is time to join the multicast group using the `join()` method as follows. Note that you must use a multicast IP address for the group.

```
String MULTICAST_IP = "239.1.1.1";

// Join the multicast group on interf interface
InetAddress group = InetAddress.getByName(MULTICAST_IP);
MembershipKey key = client.join(group, interf);
```

The `join()` method returns an object of the `MembershipKey` class that represents the membership of the datagram channel with the multicast group. If a datagram channel is not interested in receiving multicast datagrams anymore, it can use the `drop()` method of the `key` to drop its membership from the multicast group.

TIP
A datagram channel may decide to receive multicast datagrams only from selective sources. You can use the `block(InetAddress source)` method of the `MembershipKey` class to block a multicast datagram from the specified `source` address. Its `unblock(InetAddress source)` lets you unblock a previously blocked source address.

At this point, receiving datagrams that are addressed to the multicast group is just a matter of calling the `receive()` method on the channel as shown below.

```
// Prepare a buffer to receive the message from the multicast group
ByteBuffer buffer = ByteBuffer.allocate(1048);

// Wait to receive a message from the multicast group
client.receive(buffer);
```

After you are done with the channel, you can drop its membership from the group as shown below.

```
// We are no longer interested in receiving multicast message from
// the group. So, we need to drop the channel's membership from the group
key.drop();
```

Finally, you need to close the channel using its `close()` method as shown below.

```
// Close the channel
client.close();
```

To send a message to a multicast group, you do not need to be a member of that multicast group. You can send a datagram to a multicast group using the `send()` method of the `DatagramChannel` class.

Listing 3-20 has a program that joins a multicast group as a member. It waits for a message from a multicast group to arrive; prints the message and quits. Listing 3-21 has a program that sends a message to a multicast group. You can run multiple instances of the `DGCMulticastClient` class, and then, run the `DGCMulticastServer` class. All client instances should receive and print the same message on the standard output.

Listing 3-20: A DatagramChannel based multicast client program that waits for a message from a multicast group

```
// DGCMulticastClient.java
package com.jdojo.chapter3;

import java.io.IOException;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.NetworkInterface;
import java.net.StandardProtocolFamily;
import java.net.StandardSocketOptions;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;
import java.nio.channels.MembershipKey;

public class DGCMulticastClient {
     public static void main(String[] args) {
          String MULTICAST_IP = "239.1.1.1";
          int MULTICAST_PORT = 8989;

          DatagramChannel client = null;
          MembershipKey key = null;

          try {

               // Get the reference of a network interface
               NetworkInterface interf =
                                   NetworkInterface.getByName("eth0");

               // Create, configure and bind the client datagram channel
               client = DatagramChannel.open(StandardProtocolFamily.INET);
               client.setOption(StandardSocketOptions.SO_REUSEADDR, true);
               client.bind(new InetSocketAddress(MULTICAST_PORT));
               client.setOption(StandardSocketOptions.IP_MULTICAST_IF,
                                interf);

               // Join the multicast group on the interf interface
               InetAddress group = InetAddress.getByName(MULTICAST_IP);
               key = client.join(group, interf);
```

```
                // Print some useful messages for the user
                System.out.println("Joined the multicast group:" + key);

                System.out.println("Waiting for a message from the" +
                                   " multicast group....");

                // Prepare a data buffer to receive a message from
                // the multicast group
                ByteBuffer buffer = ByteBuffer.allocate(1048);

                // Wait to receive a message from the multicast group
                client.receive(buffer);

                // Convert the message in the ByteBuffer into a string
                buffer.flip();
                int limits = buffer.limit();
                byte bytes[] = new byte[limits];
                buffer.get(bytes, 0, limits);
                String msg = new String(bytes);

                System.out.format("Multicast Message:%s%n", msg);
            }
            catch (IOException  e) {
                e.printStackTrace();
            }
            finally {
                // Drop the membership from the multicast group
                if (key != null) {
                    key.drop();
                }

                // Close the client channel
                if (client != null) {
                    try {
                        client.close();
                    }
                    catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
}
```

Listing 3-21: A DatagramChannel based multicast program that sends a message to a multicast group

```
// DGCMulticastServer.java
package com.jdojo.chapter3;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.NetworkInterface;
import java.net.StandardSocketOptions;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;
```

```
public class DGCMulticastServer {
     public static void main(String[] args) {
          String MULTICAST_IP = "239.1.1.1";
          int MULTICAST_PORT = 8989;

          DatagramChannel server = null;
          try {
               // Get a datagram channel object to act as a server
               server = DatagramChannel.open();

               // Bind the server to any available local address
               server.bind(null);

               // Set the network interface for outgoing multicast data
               NetworkInterface interf =
                                   NetworkInterface.getByName("eth0");

               server.setOption(StandardSocketOptions.IP_MULTICAST_IF,
                                interf);

               // Prepare a message to send to the multicast group
               String msg = "Hello from multicast!";
               ByteBuffer buffer = ByteBuffer.wrap(msg.getBytes());

               // Get the multicast group reference to send data to
               InetSocketAddress group =
                    new InetSocketAddress(MULTICAST_IP, MULTICAST_PORT);

               // Send the message to the multicast group
               server.send(buffer, group);
          }
          catch (IOException e) {
               e.printStackTrace();
          }
          finally {
               try {
                    server.close();
               }
               catch (IOException ex) {
                    ex.printStackTrace();
               }
          }
     }
}
```

Further Reading

Network programming in Java is a vast topic. There are a few books written especially on this topic. This chapter covers only the basics of network programming support that is available in Java, Java also supports secured socket communications using a Secured Socket Layer (SSL) protocol. The classes for secured socket communication programming are in the `javax.net.ssl` package. This chapter does not cover sockets based on SSL. We have not covered many of the options for sockets that you can use in your Java programs. If you want to do advance level network

programming in Java, it is recommended that you read a book that devotes itself solely on network programming in Java after you finish this chapter.

Chapter 6. Java Native Interface

What is Java Native Interface?

Java Native Interface (JNI) is a programming interface that facilitates interaction between Java code and the code written in native languages, e.g., C, C++, FORTRAN, etc. JNI supports calling C and C++ functions directly from Java. If you need to use native code written in any other language, e.g., FORTRAN, you can use a C/C++ wrapper function to call it from Java. Interaction can take place both ways. Java code can call native code and vice versa as shown in Figure 6-1.

Figure 6-1: The JNI architecture

Java calls native code using native methods. A native method in Java context is a method, which is declared in Java and implemented in native language such as C/C++. The native method implementation is compiled into a *shared library*, which is loaded by the JVM. A shared library is called a dynamic link library (`DLL`) on Win32, and a shared object (SO) on UNIX. In Java code, you call a Java method and a native method the same way. A Java program is compiled into platform-independent format called byte codes. Native code is compiled into a platform-dependent format. Therefore, if a Java application uses native code, it is no longer portable to other platforms unless you develop the same shared library on all platforms. Sometimes, you may access platform specific features inside the native code, which is used from Java application and, in that case, you should be aware that your Java application cannot be run on other platforms.

Why would someone use JNI when Java provides a rich set of features through its class library? There may be occasions when it is necessary to use JNI to access native code in Java for the following reasons.

- If a Java application needs to implement some platform-specific features, which are not possible to implement using Java API.
- You may already have legacy code written in native languages and you want to reuse it in your Java application.
- You are developing a time-critical Java application where Java code does not perform as fast as expected. You can move the time-critical section of your Java code to native code.

TIP
You should consider using JNI in a Java application as a last resort. You must explore all possibilities of implementing the needed features using Java API. Using JNI also changes the skill set that is required to develop an application. Either the developers who are working on Java application are trained in the native language (C/C++) or new developers are brought into the team who know the native language. Using native code in a Java application makes the application less stable and prone to security risks as the native code is run outside the JVM.

We will use C++ to implement native methods in this chapter. You can use C language instead. All code examples in C++ listed in this chapter can be moved to C language with minor changes. We will specify the differences between C++ code and C code whenever you need to make changes in C++ codes to convert them to C.

System Requirements

You need a C or C++ compiler that can create a shared library. You also need a JDK installed on your computer to generate C/C++ header files. The native code referenced in this chapter has been developed using Visual Studio 2005 on a Windows platform. Java 6 was used to compile and run the Java code. However, having Visual Studio is not a requirement to run any examples. All you need is a C/C++ compiler to create a shared library on your platform.

Getting Started with JNI

This section describes the steps to develop native methods and use them in Java. We will use Visual Studio 2005 to write our C++ code to implement native methods. You do not have to use any specific C++ editor such as Visual Studio 2005 to write the C++ code. You can use any text editor, for example, notepad on Windows and `vi` editor on UNIX, to write the code. All you need is a C/C++ compiler that will compile your C/C++ code into a shared library (a `.dll` file on Win32 and `.so` file on UNIX). You can also use a command-line version of a C/C++ compiler to create a shared library if you are not using a C/C++ editor. Developing a Java application that uses JNI involves the following steps. We will discuss each step in detail.

- Writing Java Program
- Compiling Java Program
- Creating C/C++ Header File
- Writing C/C++ Program
- Creating Shared Library
- Running Java Program

Writing Java Program

A Java program that uses JNI differs from a Java-only program only in two things: loading the shared library and declaring the native method. The shared library that contains the native method implementation must be loaded before Java can call the native method. A shared library is loaded using the `load(String libraryNameWithoutExtension) static` method of the `java.lang.System` class as shown below.

```
// Load a shared library named hellojni
System.loadLibrary("hellojni");
```

You can also load a shared library using the `loadLibrary()` method of the `java.lang.Runtime` class. Internally, the `loadLibrary()` method of the `System` class calls the `loadLibrary()` method of the `Runtime` class. The above code can be rewritten as follows.

```
// Load the shared library
Runtime.getRuntime().loadLibrary("hellojni");
```

Note that you need to pass a shared library name without the file extension to the `loadLibrary()` method. For example, if your shared library file name is `hellojni.dll` on Windows or `hellojni.so` on UNIX, you need to use `hellojni` as the shared library name. The `loadLibrary()` method will append the file extension to find the shared library. This way, you do not need to change your Java code, which loads the shared library if you intend to run it on different platforms. As long as the shared library name is the same on all platforms your Java code will run without any changes.

You can also load a shared library using the `load()` method of the `System` or `Runtime` class. The `load()` method accepts the absolute path of a shared library with the file extension. Suppose a `hellojni.dll` file on Windows platforms is in the `C:\myjni` directory, a call to the `load()` method will look as follows:

```
// Load the shared library
System.load("C:\\myjni\\jnitest.dll");
```

Note that using the `load()` method forces you to use the absolute path and the file extension of the shared library, which makes your Java code non-portable to other platforms. We will use the `loadLibrary()` method of the `System` class to load shared library in our examples in this chapter.

How does the `loadLibrary()` method find the shared library file in the file system by just knowing the library name? You have two ways to let the JVM know about the location of your shared library.

- Include the directory that contains the shared library into the `PATH` environment variable on Windows and `LD_LIBRARY_PATH` environment variable on UNIX.
- Specify the directory (or directories separated by semi-colon), which contains the shared library using the `java.library.path` JVM property as a command line option. The following command assumes that the `hellojni` shared library is placed in the `C:\myjni\lib` directory.

  ```
  java -Djava.library.path=C:\myjni\lib your-class-name-to-run
  ```

A native method that is used in Java does not have a body written in Java, because its implementation exists in the native code. However, you need to declare the native method in Java before you can use it. It is declared using the `native` keyword. A `native` method declaration in Java code ends with a semi-colon. The following snippet of code declares a `native` method named `hello()`, which has no parameters and returns `void`.

```
public class XYZ {
     // Declare a native method hello()
     public native void hello();
}
```

Calling a native method in Java code is the same as calling any other Java methods.

```
XYZ xyz = new XYZ();
xyz.hello();
```

You can declare a native method to have `public`, `private`, `protected`, or package-level scope. A `native` method can be declared `static` or non-static. You can have as many `native` methods in a Java class as you want.

You cannot declare a `native` method as `abstract`. This implies that an interface cannot have a `native` method, because all methods declared in an interface are always `abstract`. An `abstract` method means that the method's implementation is missing and it will be implemented in Java, whereas `native` method means that the method's implementation is missing and it is implemented in native code. Declaring a method as `native` and `abstract` at the same time will be confusing as to where to look for the implementation of the method – in the Java code or in the native code. This is the reason why a method declaration cannot use the combination of the two modifiers - `abstract` and `native`.

The `native` keyword must be used only to declare methods. You cannot declare a field `native`. The following snippet of code declares two classes - `ABC` and `WontCompile`. The class `ABC` contains a valid usage of the `native` keyword, whereas the class `WontCompile` demonstrates its invalid usage.

```
public class ABC {
     public native void m1();
     private native void m2();
     protected native void m3();
     native void m4();

     public static native void m5();

     public native int m6(String str);

     // A non-native method (Java-only method)
     public int add(int a, int b) {
          return a + b;
     }
}

// Sample of Illegal use of native keyword in a Java class
public class WontCompile {
     private native String name; // a field cannot be native

     // A method cannot be abstract as well as native
     public abstract native String getName();
}
```

Now, we are ready to write Java code to call our first native method. We will name our native method as `hello()`. It does not accept any parameters and does not return any value. We will implement it in C++ later and it will print a message, `Hello JNI`, on the standard output. Listing 6-1 has the complete code for the `HelloJNI` class. It performs three things.

- It loads a `hellojni` shared library in its `static` initializer. Note that you do not need to have the `hellojni` shared library (`hellojni.dll` on Windows and `hellojni.so` on UNIX) when you write and compile the `HelloJNI` class. The shared library is required when you run the `HelloJNI` class.

  ```
  static {
      System.loadLibrary("hellojni");
  }
  ```

- It declares a `native` method named `hello()`, which will be implemented in C++ code later. Note that the Java compiler will let you compile the `HelloJNI` class with the `hello()`

`native` method declaration without having the native code that implements the `hello()` method. The implementation of `hello() native` method will be required when the `hello()` native method is called at runtime.

```
public native void hello();
```

- It creates an object of the `HelloJNI` class in its `main()` method and calls its `hello()` method.

```
HelloJNI helloJNI = new HelloJNI();
helloJNI.hello();
```

The code for the `HelloJNI` class is simple enough to follow. There is nothing extraordinary that we have to do inside the Java code to use a `native` method. You cannot run this class yet, because when you run it, it will look for a `hellojni` shared library with the native code for the `hello()` method, which we have not written yet.

Listing 6-1: A HelloJNI class that uses a native method hello()

```
// HelloJNI.java
package com.jdojo.chapter6;

public class HelloJNI {
     static {
          // Load the shared library using its name only
          System.loadLibrary("hellojni");
     }

     // Declare the native method
     public native void hello();

     public static void main(String[] args) {
          // Create a HelloJNI object
          HelloJNI helloJNI = new HelloJNI();

          // Call the native method
          helloJNI.hello();
     }
}
```

Compiling the Java Program

Compiling a Java program that uses native methods is the same as compiling any other Java programs. There is no special setting that you need to apply when you compile the `HelloJNI` class. You can compile it using `javac` command as usual.

```
javac HelloJNI.java
```

The above command will produce a `HelloJNI.class` file, which will contain the class definition of the `HelloJNI` class, whose fully qualified name is `com.jdojo.chapter6.HelloJNI`. Make sure that you have the `HelloJNI.class` file available, because it is necessary to perform the next step.

Creating a C/C++ Header File

Before we start writing the code for a native method in C/C++, we need to generate a header file that will contain our native method declaration in C/C++. We will use this header file when we write the implementation of our `hello() native` method. The method signature for the `hello()` method in Java and C/C++ differ significantly. You do not need to worry about the details about how to write the signature of a method in C/C++, which will be used by the Java code. Standard JDK installation provides a tool that will generate all required header files for you. The tool that you need to use is called `javah`. It is located in the `JAVA_HOME\bin` folder, where `JAVA_HOME` is the installation folder for a JDK. For example, if you have installed a JDK in `C:\java2` directory, you can find the `javah` tool in the `C:\java2\bin` folder. The `javah` tool accepts the fully qualified class name and it generates a header file with extension `.h` that contains the method signature for all `native` methods declarations in the specified class. The following command will generate a C/C++ header file for all `native` methods declarations in the `HelloJNI` class.

```
javah com.jdojo.chapter6.HelloJNI
```

The `javah` tool will look for the `HelloJNI` class in the `CLASSPATH`. If it is not in the `CLASSPATH`, you can specify the new `CLASSPATH` using a `-classpath` command-line option as follows:

```
javah -classpath=your-classpath-goes-here com.jdojo.chapter6.HelloJNI
```

The above command will generate a header file named `com_jdojo_chapter6_HelloJNI.h` and place it in the current directory. By default, the generated file name is based on the fully-qualified name of the class. A dot in the class name is replaced by an underscore and the file has a `.h` extension. You can also specify the header file name that the `javah` command will generate by using a `-o` option. You can look at other options supported by the `javah` command by executing a "`javah -help`" command.

The content of `com_jdojo_chapter6_HelloJNI.h` file, which is generated by the `javah` command, is as follows:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_jdojo_chapter6_HelloJNI */

#ifndef _Included_com_jdojo_chapter6_HelloJNI
#define _Included_com_jdojo_chapter6_HelloJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_jdojo_chapter6_HelloJNI
 * Method:    hello
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_jdojo_chapter6_HelloJNI_hello
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

You do not need to worry about the details in this header file. We only need the method signature that is generated for our `native hello()` method. Our "`void hello()`" method signature declaration in the Java code has been translated into the following method signature for the C./C++ code.

```
JNIEXPORT void JNICALL Java_com_jdojo_chapter6_HelloJNI_hello(JNIEnv *,
                                                              jobject);
```

`JNIEXPORT` and `JNICALL` are two macros. The `void` keyword denotes that our `native` method does not return any value. The `javah` command uses a rule to generate the name of the `native` method in the header file. In our case, the method name is `Java_com_jdojo_chapter6_HelloJNI_hello`. We will look at the naming rules used by the `javah` tool for native methods later. Although the method declaration of the `hello()` method in the Java code does not accept any parameters, our `native` method declaration in the header file accepts two parameters. Take it as a rule that all native method declarations in a native language will accept two additional parameters than the ones declared in the Java code. The additional parameters are added as the first and second parameters for the method in a native language. The first parameter is a pointer to a `JNIEnv` type object, which is basically a table of function pointers to facilitate interaction between the native and Java environments. The second parameter is of type either `jobject` or `jclass`. If the `native` method is declared non-static in the Java code, the second parameter is of type `jobject`, which is a reference to the Java object on which the native method is called. It is similar to the `this` reference that you have inside every non-static method in Java (or C++ methods). Since our native `hello()` method in Java has been declared non-static, the second parameter type is of `jobject`. If the native method is declared as a `static` method in Java, the second parameter will be of type `jclass` and it will be the reference to the class object in the JVM on which the `native` method is called.

At the end of this step, you should have a header file named `com_jdojo_chapter6_HelloJNI.h` with the above contents in it.

Writing the C/C++ Program

Listing 6-2 shows the C/C++ code that we need to write for our `hello()` native method. The next section describes the step-by-step process to setup a project and write the C++ code using Visual Studio 2005 on a Windows platform. We have named our file `hellojni.cpp`. In this case, the code will be the same if you choose to use the C language instead. Note that `hello` is the name of our native method in Java code, whereas in C/C++ it is named `Java_com_jdojo_chapter6_HelloJNI_hello`.

Listing 6-2: A C/C++ implementation for the hello() native method

```
// hellojni.cpp
#include <stdio.h>
#include <jni.h>
#include "com_jdojo_chapter6_HelloJNI.h"

JNIEXPORT void JNICALL Java_com_jdojo_chapter6_HelloJNI_hello
(JNIEnv *env, jobject obj) {
     printf("Hello JNI\n");
     return;
}
```

Here are the things that this program does. It uses three C/C++ compiler preprocessor `include` directives to include three header files - `stdio.h`, `jni.h` and `com_jdojo_chapter6_HelloJNI.h`. It includes `stdio.h` to use the standard Input/Output functionalities. It includes `jni.h` to use JNI related functionalities. It includes `com_jdojo_chapter6_HelloJNI.h` to include functionalities related to our new `hello()` `native` method.

The `jni.h` file is copied to `JAVA_HOME\include` when you install a JDK. For example, if you installed JDK in `C:\java2`, a `jni.h` file is copied to the `C:\java2\include` directory. There is another directory that is created under `HAVA_HOME\include` directory .The directory name is platform-dependent. It is named `win32` on Windows and `solaris` on Solaris. You need to use these two directories (e.g. `C:\java2\include` and `C:\java2\include\win32` on Windows) as an include-path option when you compile the `hellojni.cpp` file.

The `com_jdojo_chapter6_HelloJNI.h` is the file that is generated using the `javah` command (refer to the previous section for details). You can place this file in any directory on your machine. You will need to include the directory which contains this file in the include-path option when you compile the `hellojni.cpp` file.

The function signature is copied from the `com_jdojo_chapter6_HelloJNI.h` header file. Note that we have named the two parameters as `env` and `obj`. It does not matter what name you use for these parameters in your code.

```
JNIEXPORT void JNICALL Java_com_jdojo_chapter6_HelloJNI_hello
(JNIEnv *env, jobject obj)
```

We have provided the implementation for our `native` method and we have only two statements. The first statement uses the `printf()` function to print a message - `"Hello JNI"`, on the standard output, and the second one returns from the function.

```
printf("Hello JNI\n");
return;
```

Creating a Shared Library

Compile the `hellojni.cpp` file into a shared library named `hellojni`. The shared library will be a file named `hellojni.dll` on Windows and `hellojni.so` on UNIX. Your operating system may use a different file extension for a shared library. Many compilers are available that can be used to create a shared library from C/C++ code. The next section describes how to create a shared library (a DLL) on Windows platform using Visual Studio 2005. You can use any other C/C++ compiler that can create a shared library for you.

Creating a Shared Library Using Visual Studio 2005

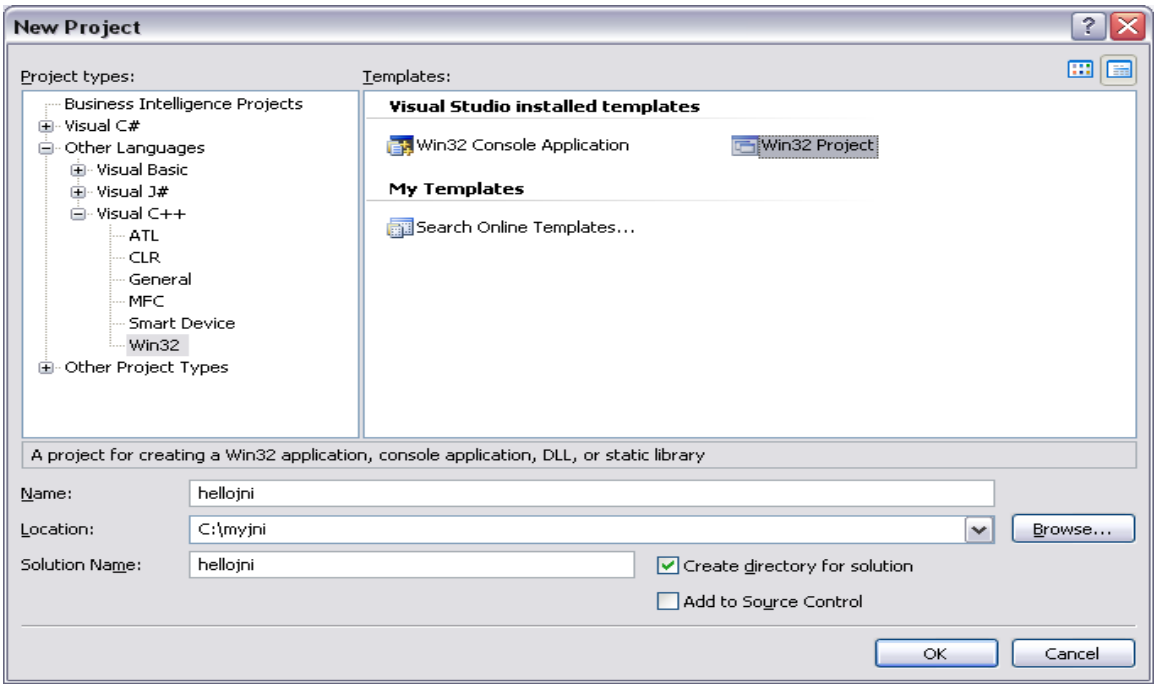
This section describes the steps to create and compile a Visual C++ Win32 project using the Microsoft Visual Studio 2005.

Step 1: Creating a Visual C++ Project

Create a new Visual C++ project using `File >> New` menu item and enter the details on New Project dialog box as shown in Figure 6-2.

- Select “Win32” under Visual C++ as “Project types”.
- Select “Win32 Project” under “Project templates”.
- Enter `hellojni` as the Name of the project.
- Enter `C:\myjni` as the Location of the project.
- Leave all other selections/values to their defaults.
- Press OK button. This will take you to the next dialog box that will display the overview of the current project settings. Press Next button on this dialog box, which will display “Application Settings” dialog box. You need to select the DLL radio button option under the “Application type:” and “Empty project” checkbox under the “Additional options” on this dialog box.
- On “Application Settings” dialog box, select DLL as “Application Type” and Empty Project as “Additional Options”. Press the Finish button. Now, you have an empty Visual C++ project.

Figure 6-2: Entering VC++ project details



We need to add a new C++ source file to our project to write C++ code. Select “Source Files” tree node in the Solution explorer and select the `Add >> New Item` from the right click menu options as shown in Figure 6-3. You will get the “Add New Item” dialog box as shown in Figure 6-4. Select “Code” under Categories and “C++ File (.cpp) under templates. Enter hellojni as the Name of the file. Leave all other fields to their default values.

Figure 6-3: Adding a new C++ source file to hellojni VC++ project

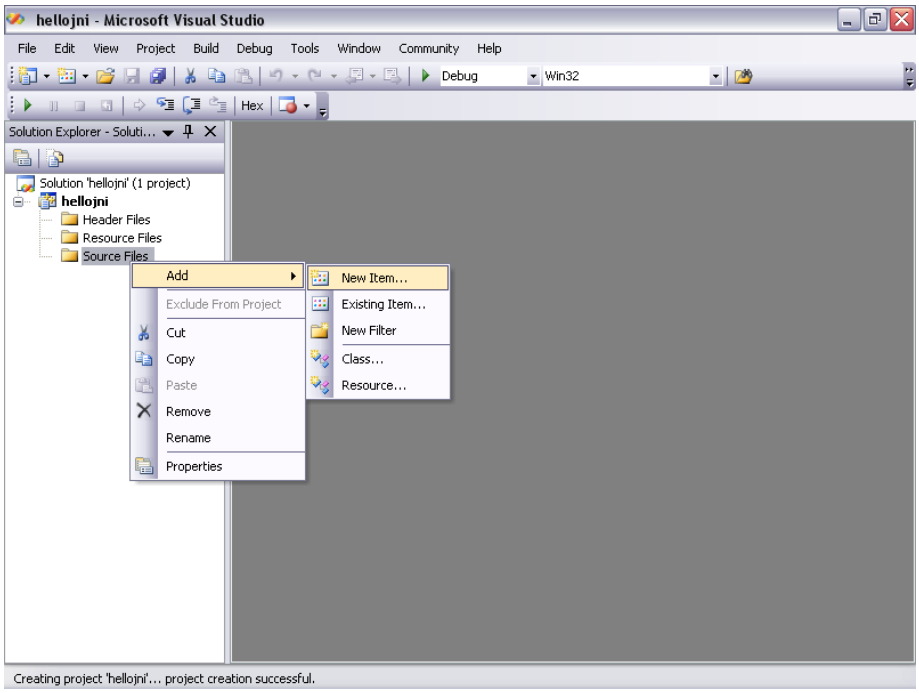
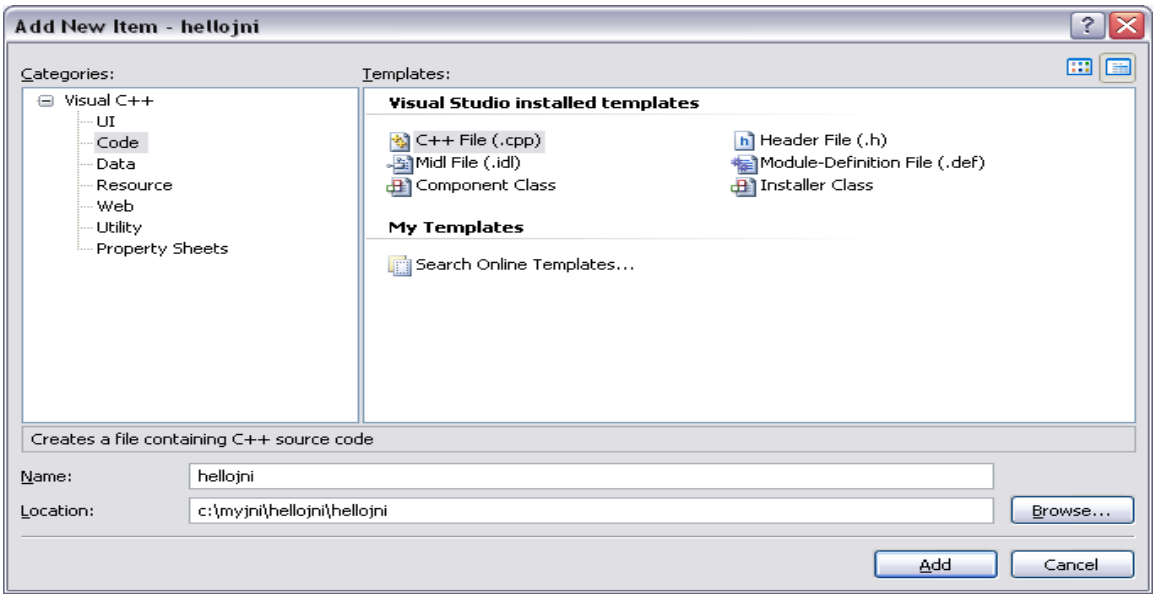


Figure 6-4: Enter C++ source file name and location



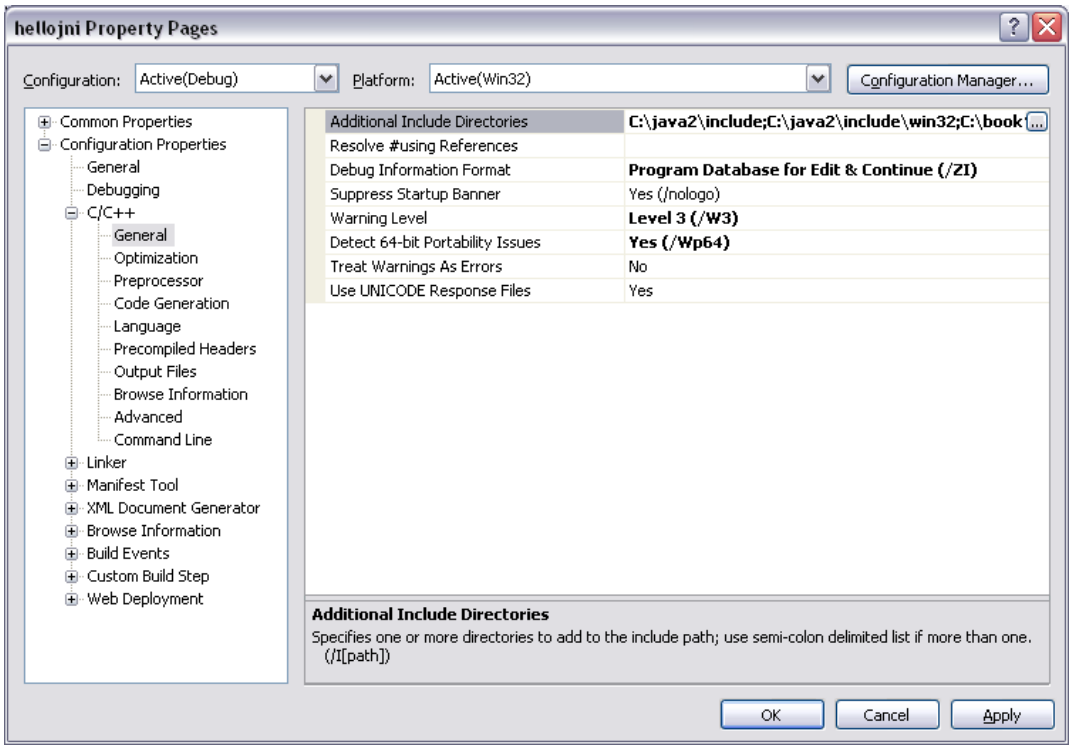
We are ready to write the C++ code for our JNI method. Open hellojni.cpp source file in the Solution Explorer and enter the following code.

```
// hellojni.cpp
#include <stdio.h>
#include <jni.h>
```

```
#include "com_jdojo_chapter6_HelloJNI.h"

JNIEXPORT void JNICALL Java_com_jdojo_chapter6_HelloJNI_hello
(JNIEnv *env, jobject obj) {
     printf("Hello JNI\n");
     return;
}
```

It is time to compile the hellojni.cpp file to create a DLL. We need to set some properties for our project before we can compile our hellojni.cpp file. We need to tell the VC++ project about the location of the header files we need to include in our program. Open the "Property Pages" dialog box for the project. You can open the "Properties Pages" dialog box by right clicking on the project name in the Solution Explorer and selecting "Properties" menu item or by selecting `Project >> hellojni properties` item from the main menu, which will display a dialog box as shown in Figure 6-5.

Figure 6-5: Setting the Include directory paths for VC++ project

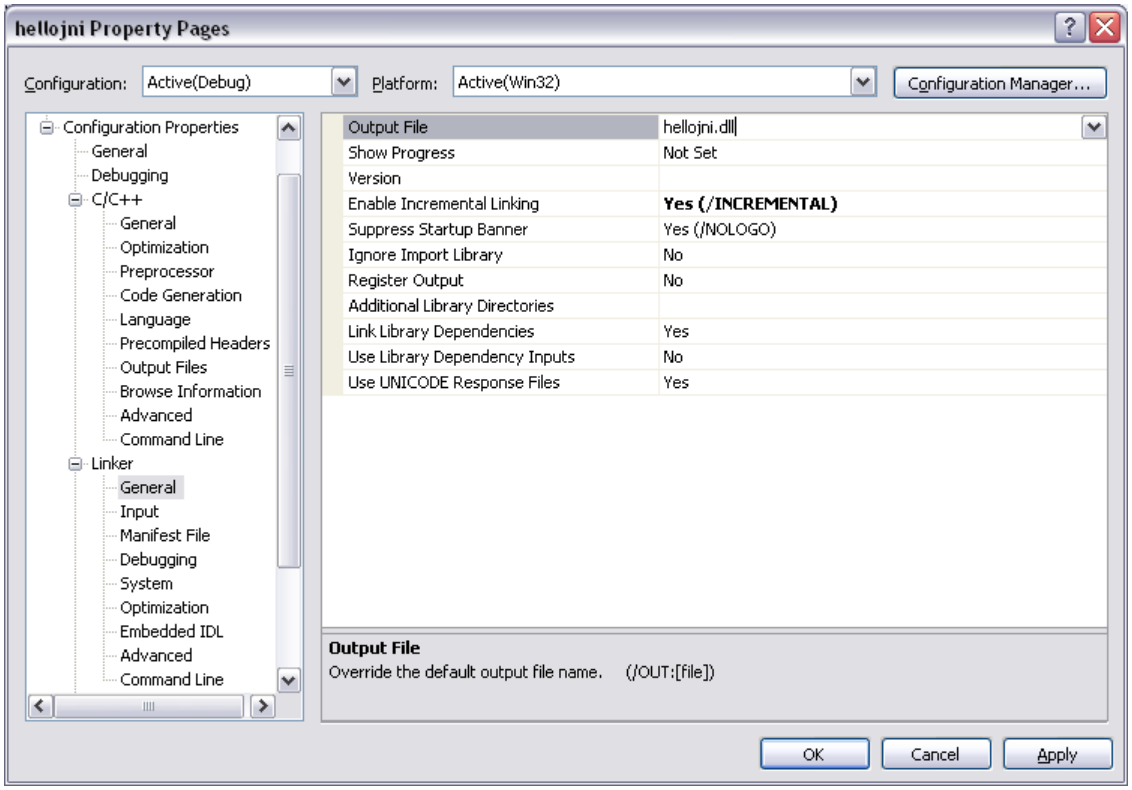


We need to set the include-path for `jni.h` (and other header files) that is used in our C++ program. Select the General tree node under C/C++ tree node and enter the following value for "Additional Include Directories" property, where `C:\java2` is the JDK installation directory. If you have installed the JDK in some other directory, replace `C:\java2` by that directory path. In the following path list, `"C:\book\myheaderfiles"` denotes the directory where we have stored our `com_jdojo_chapter6_HelloJNI.h` header file. Replace this directory with your directory where you have stored the `com_jdojo_chapter6_HelloJNI.h` header file. The main point of setting this property is to let the VC++ compiler know about the list of directories where it can search for the `.h` header files that are included in the C++ source file.

```
C:\java2\include;C:\java2\include\win32;C:\book\myheaderfiles
```

By default, VC++ generates a DLL file name based on the VC++ project name. Since our project name is `hellojni`, it will generate the `hellojni.dll` file when we build our project. You can change the default name of the DLL file by setting the "Output File" property under `Linker >> General` node as shown in Figure 6-6.

Figure 6-6: Setting DLL file name for VC++ project



Build the project by pressing `F6` or selecting `Build >> Build Solution` from the main menu. This will generate a `hellojni.dll` file in the `C:\myjni\hellojni\hellojni` directory, which is your shared library on Windows platform.

Running the Java Program

We are ready to run our `HelloJNI` Java class. Before you can run this class, make sure that you have created the `hellojni` shared library (e.g. a `hellojni.dll` file on Windows) successfully. Suppose you have placed the `hellojni` shared library file in the `C:\myjni\lib` directory. Run the `HelloJNI` class using the following command.

```
java -Djava.library.path=C:\myjni\lib com.jdojo.chapter6.HelloJNI
```

The "`-Djava.library.path=C:\myjni\lib`" option for the `java` command instructs the JVM to look for shared libraries in the `C:\myjni\lib` directory. If the above command runs successfully, it will print a message, "`Hello JNI`", on the standard output.

Native Function Naming Rules

The `javah` command uses a naming rule, which is based on name-mangling, to generate native method names in the C/C++ header file. Java runtime uses the same rule to resolve the Java native method name to the native function name in a shared library. The name-mangling rule is used, so that the name generated for the native function is a valid C/C++ name without a name collision. You can think of name-mangling as simply replacing invalid characters with characters, which make up a valid function name. The native function name is generated based on the following parts, which are concatenated using an underscore.

- The method name starts with the word `Java`.
- Mangled fully qualified name of the package of the Java class that contains the native method's declaration. An underscore is used as a package/sub-package separator.
- The native method name in Java
- If a native method is overloaded, two underscores followed by the mangled method's signature

Java runtime uses two names for a native function – a short name and a long name. The short name does not use two underscores followed by the mangled method's signature. Java runtime searches the shared library for the short name first. If it does not find a function using the short name, it searches the shared library with the long name.

The mangled-name uses a conversion table shown in Table 6-1. Characters such as a semi-colon and beginning with a square bracket may occur as part of a method's parameter signature that is used internally by Java. The following are four examples that show the parameter signature that is used by Java internally.

```
public static void javaPrintMsg(java.lang.String);
     Signature: (Ljava/lang/String;)V

public void javaCallBack();
     Signature: ()V

public static void main(java.lang.String[]);
     Signature: ([Ljava/lang/String;)V
```

Table 6-1: Escape sequence used in name-mangling process.

Original Character	Substituted Character
Any non-ASCII Unicode Character	`_0xxxx` Note that alphabets used in `_oxxxx` are all lowercase e.g. `_0abcd`
`_` (an underscore)	`_1`
`;` (a semi-colon)	`_2`
`[` (a beginning square bracket)	`_3`

If you declare a parameter of type `java.lang.String`, it is used internally as `Ljava/lang/String;`. To know about the signature of a method that is used internally by Java, you need to use the `javap` command with a `-s` option. For example, the following command will print the method signatures for all methods in the `com.jdojo.chapter6.HelloJNI` class. You can use a `-private` option to print signatures of all methods including the `private` ones.

```
javap -s -private com.jdojo.chapter6.HelloJNI
```

If you are required to use a method signature of a Java method inside a JNI function in native code, you should run the `javap` command to get the method signature instead of typing it by hand. You can learn the rules used to make up the method signature that is used internally by Java. However, using the `javap` command makes it easy to get the method signature of a Java method. Let us consider the declaration of some native methods in a class `Test` as follows.

```
package com.jdojo.chapter6;

public class Test {
     private native void sayHello();
     private native void printMsg(String msg);
     private native int[] increment(int[] num, int incrementValue);
     private native double myMethod(int i, String s[], String ss);
     private native double myMethod(double i, String s[], String ss);
     private native double myMethod(short i, String s[], String ss);
}
```

If you compile and run the `javah` command for the `com.jadojo.chapter6.Test` class, you get the following header file. You can look at the native function names that are generated for different `native` method's declarations. Do not worry about the data types used for the function's parameters. We will cover data type mapping between Java and native language in the next section.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_jdojo_chapter6_Test */

#ifndef _Included_com_jdojo_chapter6_Test
#define _Included_com_jdojo_chapter6_Test
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_jdojo_chapter6_Test
 * Method:    sayHello
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_jdojo_chapter6_Test_sayHello
  (JNIEnv *, jobject);

/*
 * Class:     com_jdojo_chapter6_Test
 * Method:    printMsg
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_com_jdojo_chapter6_Test_printMsg
  (JNIEnv *, jobject, jstring);
```

```
/*
 * Class:     com_jdojo_chapter6_Test
 * Method:    increment
 * Signature: ([II)[I
 */
JNIEXPORT jintArray JNICALL Java_com_jdojo_chapter6_Test_increment
  (JNIEnv *, jobject, jintArray, jint);

/*
 * Class:     com_jdojo_chapter6_Test
 * Method:    myMethod
 * Signature: (I[Ljava/lang/String;Ljava/lang/String;)D
 */
JNIEXPORT jdouble JNICALL
Java_com_jdojo_chapter6_Test_myMethod__I_3Ljava_lang_String_2Ljava_lang_S
tring_2
  (JNIEnv *, jobject, jint, jobjectArray, jstring);

/*
 * Class:     com_jdojo_chapter6_Test
 * Method:    myMethod
 * Signature: (D[Ljava/lang/String;Ljava/lang/String;)D
 */
JNIEXPORT jdouble JNICALL
Java_com_jdojo_chapter6_Test_myMethod__D_3Ljava_lang_String_2Ljava_lang_S
tring_2
  (JNIEnv *, jobject, jdouble, jobjectArray, jstring);

/*
 * Class:     com_jdojo_chapter6_Test
 * Method:    myMethod
 * Signature: (S[Ljava/lang/String;Ljava/lang/String;)D
 */
JNIEXPORT jdouble JNICALL
Java_com_jdojo_chapter6_Test_myMethod__S_3Ljava_lang_String_2Ljava_lang_S
tring_2
  (JNIEnv *, jobject, jshort, jobjectArray, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

Data Type Mapping

JNI defines mapping between data types used in Java and native functions. Table 6-2 lists the mapping for primitive data types between Java and native C/C++ language. Note that all you have to do is to add a `j` in front of the name of a primitive data type in Java and you get the equivalent data type name in C/C++. JNI also defines a data type named `jsize`, which is used to store the length, for example, length of an array or a string.

Table 6-2: Mapping between Java primitive data types and JNI native data types

Java Primitive Types	Native Primitive Type	Description
`boolean`	`jboolean`	unsigned 8 bits
`byte`	`jbyte`	signed 8 bits
`char`	`jchar`	unsigned 16 bits
`double`	`jdouble`	64 bits
`float`	`jfloat`	32 bits
`int`	`jint`	signed 32 bits
`long`	`jlong`	signed 64 bits
`short`	`jshort`	signed 16 bits
`void`	`void`	N/A

JNI defines reference type equivalents for Java reference types. It is not possible to define a separate type in the JNI for all reference types that can be created in Java. All Java reference types can be mapped to a `jobject` JNI reference type. We have some specialized JNI reference types that represent commonly used reference types in Java for example `jstring` is used in the JNI to represent a `java.lang.String` in Java. Table 6-3 lists the reference type mapping between Java and the JNI.

Table 6-3: Reference type mapping between Java and JNI

Java Reference Type	JNI Type
Any Java object	`jobject`
`java.lang.String`	`jstring`
`java.lang.Class`	`jclass`
`java.lang.Throwable`	`jthrowable`

The JNI defines separate reference types to represent Java arrays. It defines a `jarray` type that represents any Java array type. It has a specialized array type for each type of array in Java. In JNI, an array type is named like `jXXXArray`, where `XXX` could be `object`, `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`. For example, `jintArray` in C/C++ represents an `int` array in Java. Note that all reference type arrays in Java are represented by `jobjectArray` type in C/C++.

While working with C/C++ code using JNI, you will come across another type called `jvalue`. It is a union type defined in C/C++ as shown below.

```
typedef union jvalue {
     jboolean z;
     jbyte    b;
     jchar    c;
     jshort   s;
     jint     i;
     jlong    j;
     jfloat   f;
     jdouble  d;
     jobject  l;
```

```
} jvalue
```

Note that the `jvalue` union type does not have an equivalent type in Java. Typically, the `jvalue` type is defined as a parameter type in built-in functions that are part of the JNI API.

Using JNI Functions in C/C++

JNI functions let you access the JVM data structures and objects in native codes. Sometimes, they let you convert the data in a particular format that is passed between Java and native environments. All native functions have their first parameter, which is always a pointer to `JNIEnv`, which in turn is a pointer to a table of all JNI function pointers.

There are always two versions of functions that you can call on type `JNIEnv` – one for C and one for C++. The C version of the function accepts a pointer to `JNIEnv` as the first parameter and C++ will not have that first parameter. The two versions of the same methods, C and C++, are called differently. The following snippet of code shows the difference in calling a JNI function in C and C++ assuming `FuncXxx` is the function name and `env` is a pointer to `JNIEnv` type.

```
(*env)->FuncXxx(env, list-of-arguments...); // C style
env->FuncXxx(list-of-arguments...);         // C++ style
```

This chapter uses C++ way of calling JNI functions. You can convert the code to C style easily by using the above snippet of code as a reference.

As a concrete example, the following are the function signatures for `GetStringUTFChars()` JNI function that lets you convert a Java string to a UTF-8 string format.

C Version of the GetStringUTFChars() JNI function

```
const char * GetStringUTFChars(JNIEnv *env, jstring string,
                               jboolean *isCopy);
```

C++ Version of the GetStringUTFChars() JNI function

```
const char * GetStringUTFChars(jstring string, jboolean *isCopy);
```

If you want to call this function in C or C++, your codes will look as follows.

```
// C Code
const char *utfMsg = (*env)->GetStringUTFChars(env, msg, iscopy);

// C++ Code
const char *utfMsg = env->GetStringUTFChars(msg, iscopy);
```

Working with Strings

Strings are represented differently in Java and C/C++. In Java, a string is represented as a sequence of 16-bit Unicode characters, whereas in C/C++ a string is a pointer to a sequence of null-terminated characters. The `jstring` reference type that you can use in the native code

represents an instance of the `java.lang.String` class, which is a sequence of 16-bit Unicode characters. JNI has functions that let you convert a Java string to a native string and vice-versa. One set of string functions works with UTF-8 strings, whereas the other set works with Unicode strings. When you receive a string in native code from Java, you must convert it to native format (UTF-8 or Unicode) before using it. The same logic goes for returning a string from native code to Java. You must convert the native string to an instance of `jstring` before it can be returned to Java.

Let us start with an example in which we will pass a string from Java code to C/C++ code. The C/C++ code will convert the Java string to a native UTF-8 format and print it on the standard output using the `printf()` function. The native methods declaration in Java would be as follows.

```
public native void printMsg(String msg);
public native String getMsg();
```

The `printMsg()` method accepts a Java string and its native function will print it on the standard output. The `getMsg()` method returns a native string to Java and Java will print it on the standard output. Listing 6-3 contains the Java code that declares these two native methods.

Listing 6-3: Passing string from Java to native function and vice-versa

```
// JNIStringTest.java
package com.jdojo.chapter6;

public class JNIStringTest {
     static {
          System.loadLibrary("jnistring");
     }

     public native void printMsg(String msg);
     public native String getMsg();

     public static void main(String[] args) {
          JNIStringTest stringTest = new JNIStringTest();

          String javaMsg = "Hello from Java to JNI";
          stringTest.printMsg(javaMsg);

          String nativeMsg = stringTest.getMsg();
          System.out.println(nativeMsg);
     }
}
```

The following are the native function declarations for `printMsg()` and `getMsg()` in C/C++. Note that the first two parameters to the native functions are of type `JNIEnv` and `jobject`. The `printMsg()` function has a third parameter of type `jstring` and its return type is `void`. The `getMsg()` function has only two standard parameters and it returns a `jstring`.

```
JNIEXPORT void JNICALL Java_com_jdojo_chapter6_JNIStringTest_printMsg
     (JNIEnv *env, jobject obj, jstring msg);

JNIEXPORT jstring JNICALL Java_com_jdojo_chapter6_JNIStringTest_getMsg
     (JNIEnv *env, jobject obj);
```

To convert a `jstring` to a UTF-8 native string, you need to use the `GetStringUTFChars()` JNI function that you can access using a `JNIEnv` reference. The `GetStringUTFChars()` JNI function has two versions – one for C and one for C++.

The `GetStringUTFChars()` function converts Java string (in a `jstring` in C/C++ code) to a UTF-8 format and returns a pointer to the converted UTF-8 string. If it fails, it returns `NULL`. The `GetStringUTFChars()` function may have to make a copy of the original Java string object in memory for converting it to UTF-8 format. The `isCopy` parameter to the functions, which is a pointer to a `boolean` variable, can be used to check if this function had to copy the original Java string. If `isCopy` is not `NULL`, it is set to `JNI_TRUE` if a copy of the Java string was made. Otherwise, it is set to `JNI_FALSE`. Once you are done with the returned value of this function, you must call `ReleaseStringUTFChars()` method to release the memory. The C and C++ style signatures of this `method are as follows.`

```
// C Style
void ReleaseStringUTFChars(JNIEnv *env, jstring string, const char *utf);

// C++ Style
void ReleaseStringUTFChars(jstring string, const char *utf);
```

Listing 6-4 has implementations for `printMsg()` and `getMsg()` native methods in C++. The code for `getMsg()` is simple. It uses a `NewStringUTF()` JNI function to get a Java string from the native string, which is a pointer to characters. You can compile the C++ implementations of the native functions – `printMsg()` and `getMsg()` into a `jnistring` shared library and run the `JNIStringTest` class listed in Listing 6-3 to see the result.

Listing 6-4: Implementation of the printMsg() and getMsg() methods in C++

```
JNIEXPORT void JNICALL Java_com_jdojo_chapter6_JNIStringTest_printMsg
(JNIEnv *env, jobject obj, jstring msg) {
     const char *utfMsg;
     jboolean *iscopy = NULL;

     // Get the UTF string
     utfMsg = env->GetStringUTFChars(msg, iscopy);
     if (utfMsg == NULL) {
          printf("Could not convert Java string to UTF-8 string.\n");
          return ;
     }

     // Print the message on the standard output
     printf("%s\n", utfMsg);

     // Release the memory
     env->ReleaseStringUTFChars(msg, utfMsg);
}

JNIEXPORT jstring JNICALL Java_com_jdojo_chapter6_JNIStringTest_getMsg
(JNIEnv *env, jobject obj) {
     const char *utfMsg = "Hello from JNI to Java";
     jstring javaString = env->NewStringUTF(utfMsg);
     return javaString;
}
```

TIP
You can use the `GetStringUTFLength(jstring string)` JNI function to get the length of a `jstring` in bytes to represent it in UTF-8 format. JNI also has functions that let you work with Unicode native strings. The Unicode string functions are named UTF string functions without the word "UTF". For example, to get the length of a `jstring` in terms of Unicode characters, you have a `GetStringLength()` function as opposed to the `GetStringUTFLength()` function. To construct a new Java `String` (a `jstring`) from Unicode characters, you have a `NewString()` JNI function as opposed to the `NewStringUTF()` JNI function, which creates a Java string from a UTF-8 native string. Sometimes, you may need to convert a Java `String` in `jstring` to a native encoding and vice-versa. You can use the `java.lang.String` class, which has a rich set of constructors and methods that let you convert string in one encoding to byte array and vice-versa. We will cover how to access Java classes in the native code in a later section.

Working with Arrays

JNI lets you pass an array of primitive and reference types from Java to native code and vice-versa. You cannot access or work with Java arrays directly in native code. Rather, you will need to use JNI functions to work with them. JNI has a different set of functions for primitive and reference arrays. Some functions are common to both types. All array related methods used in this section use the C++ version. Add "JNIEnv *env" as the first parameter to them to get the corresponding C version.

The `GetArrayLength()` method returns the length of an array of a primitive or reference type. Its signature is shown below.

```
jsize GetArrayLength(jarray array)
```

You can use the `New<XXX>Array()` method to create an array of a primitive type, where `<XXX>` is one of the primitive types - `Boolean`, `Byte`, `Char`, `Double`, `Float`, `Int`, `Long`, or `Short`. You need to pass the length of the primitive type array as a parameter to this method. It returns `NULL` if an array could not be created. For example, the following snippet of code creates an `int` array and a `double` array each of length 10.

```
jintArray iArray = env->NewIntArray(10);
jdoubleArray dArray = env->NewDoubleArray(10);
```

You can use `Get<XXX>ArrayElements()` to get the contents of a primitive array, where `<XXX>` is one of the primitive types - `Boolean`, `Byte`, `Char`, `Double`, `Float`, `Int`, `Long`, or `Short`. Its signature is as follows. The `<RRR>` is the JNI native data type, e.g., `jint`, `jdouble` and `<AAA>` is a JNI array type, e.g., `jintArray`, `jdoubleArray`, etc.

```
<RRR> *Get<XXX>ArrayElements(<AAA> array, jboolean *isCopy)
```

The `isCopy` parameter indicates if the returned array elements are a copy of the original array. If `isCopy` is not `NULL`, it is set to `JNI_TRUE` if a copy of original array was made. It is set to `JNI_FALSE` if a copy of original array was not made. You can also make changes to the array elements in the native code that will be reflected to the original array. You need to release the elements, which you get using this method after you are done with them. You need to use the `Release>XXX>ArrayElements()` method to release the array elements, whose signature is as follows:

```
void Release<XXX>ArrayElements(<AAA> array, <RRR> *elems, jint mode)
```

The last parameter `mode` in the `Release<XXX>ArrayElements()` function indicates how the buffer, which was used in native code for array elements, is released. Its value could be 0, `JNI_COMMIT`, or `JNI_ABORT`. 0 means copy back the content and free the `elems` buffer, `JNI_COMMIT` means copy back the content, but do not free the `elems` buffer, and `JNI_ABORT` means free the buffer without copying back the possible changes. The following snippet of code accesses an `int` Java array in native code and prints all of its element values on the standard output.

```
jintArray num = get a Java array...;
const jsize count = env->GetArrayLength(num);
jboolean isCopy;
jint *intNum = env->GetIntArrayElements(num, &isCopy);

for (jsize i = 0; i < count; i++) {
     printf("%i\n", intNum[i]);
}

// Release the intNum buffer without copying back any changes
// made to the array elements
env->ReleaseIntArrayElements(num, intNum, JNI_ABORT);
```

Reference type arrays are treated differently. You can use the `NewObjectArray()` function to create a new reference type array. Note that you need to use the array element's class type object to create a reference array. The last parameter is the initial element with which all elements of the array will be initialized.

```
jobjectArray NewObjectArray(jsize length,
                            jclass elementClass,
                            jobject initialElement);
```

Unlike primitive type arrays, you do not need to get array elements for reference type arrays to access them. You can access one element at a time using the `GetObjectArrayElement()` function. You can use the `SetObjectArrayElement()` function to set the value of an array element of a reference type.

```
jobject GetObjectArrayElement(jobjectArray array, jsize index);
void SetObjectArrayElement(jobjectArray array, jsize index,
                           jobject value);
```

Let us look at examples of using arrays in a JNI application. Listing 6-5 has the Java code that declares three native methods, which uses arrays. The `sum()` method accepts an `int` array and returns the sum of all its elements as `int`. Be careful not to pass big numbers in the `int` array when you call the `sum()` method. The `concat()` method accepts a `String` array. It concatenates all elements in the array and returns a `String` object. The `increment()` method accepts an `int` array and an `int` number. It returns a new `int` array, which contains all elements of the original array, which are incremented by the specified number. The `main()` method has the code to test these three native methods. Listing 6-6 has the C++ implementation of the three native methods. The `concat()` method's implementation assumes that the length of all elements in `String` array will not exceed 500 bytes.

Listing 6-5: Example of accessing and manipulating arrays in native code

```
// JNIArrayTest.java
package com.jdojo.chapter6;

import java.util.Arrays;

public class JNIArrayTest {
     static {
          System.loadLibrary("jniarraytest");
     }

     public native int sum(int[] num);
     public native String concat(String[] str);
     public native int[] increment(int[] num, int incrementBy);

     public static void main(String[] args) {
          JNIArrayTest test = new JNIArrayTest();

          int[] num = {1, 2, 3, 4, 5};
          String[] str = {"One", "Two", "Three", "Four", "Five" } ;

          System.out.println("Original Number Array: " +
                             Arrays.toString(num));

          System.out.println("Original String Array: " +
                             Arrays.toString(str));
          int sum = 0;
          sum = test.sum(num);
          System.out.println("Sum: " + sum);

          String concatenatedStr = test.concat(str);
          System.out.println("Concatenated String: " + concatenatedStr);

          int increment = 5;
          int[] incrementedNum = test.increment(num, increment);
          System.out.println("Increment By: " + increment);
          System.out.println("Incremented Number Arrays: " +
                             Arrays.toString(incrementedNum));
     }
}
```

Listing 6-6: C++ implementation of the sum(), concat(), and increment() native methods

```
JNIEXPORT jint JNICALL Java_com_jdojo_chapter6_JNIArrayTest_sum
(JNIEnv *env, jobject obj, jintArray num) {

     jint sum = 0;
     const jsize count = env->GetArrayLength(num);

     jboolean isCopy;
     jint *intNum = env->GetIntArrayElements(num, &isCopy);

     for (jsize i = 0; i < count; i++) {
          sum += intNum[i];
     }
```

```
    // Release the intNum buffer without copying back any changes
    // made to the array elements
    env->ReleaseIntArrayElements(num, intNum, JNI_ABORT);

    return sum;
}

JNIEXPORT jstring JNICALL Java_com_jdojo_chapter6_JNIArrayTest_concat
(JNIEnv *env, jobject obj, jobjectArray strArray) {
    const int MAX_LENGTH = 500;
    char dest[MAX_LENGTH];

    for(int i = 0; i < MAX_LENGTH; i++) {
        dest[i] = NULL;
    }

    const jsize count = env->GetArrayLength(strArray);

    for (jsize i = 0; i < count; i++) {
        // Get the string object from the array
        jstring strElement =
            (jstring)env>GetObjectArrayElement(strArray, i);
        const char *tempStr = env->GetStringUTFChars(strElement, NULL);

        if (tempStr == NULL) {
            printf("Could not convert Java string to UTF-8 string.\n");
            return NULL;
        }

        // Concatenate tempStr to dest
        strcat(dest, tempStr);

        // Release the memory used by tempStr
        env->ReleaseStringUTFChars(strElement, tempStr);

        // Delete the local reference of jstring
        env->DeleteLocalRef(strElement);
    }

    jstring returnStr = env->NewStringUTF(dest);
    return returnStr;
}

JNIEXPORT jintArray JNICALL
Java_com_jdojo_chapter6_JNIArrayTest_increment
(JNIEnv *env, jobject obj, jintArray num, jint incrementBy) {

    const jsize count = env->GetArrayLength(num);

    jboolean isCopy;
    jint *intNum = env->GetIntArrayElements(num, &isCopy);

    jintArray modifiedNumArray = env->NewIntArray(count);
    jboolean isNewArrayCopy;
    jint *modifiedNumElements =
        env->GetIntArrayElements(modifiedNumArray, &isNewArrayCopy);
```

```
    for (jint i = 0; i < count; i++) {
        modifiedNumElements[i] = intNum[i] + incrementBy;
    }

    if (isCopy == JNI_TRUE) {
        env -> ReleaseIntArrayElements(num, intNum, JNI_COMMIT);
    }

    if (isNewArrayCopy == JNI_TRUE) {
        env -> ReleaseIntArrayElements(modifiedNumArray,
                                       modifiedNumElements,
                                       JNI_COMMIT);
    }

    return modifiedNumArray;
}
```

Accessing Java Objects in Native Code

You can create Java objects in native code. You can access Java objects and classes in a JVM. You can access/modify fields of a Java object inside native code. You can also invoke Java instances and `static` methods from native code.

Getting a Class Reference

An instance of the `jclass` type represents a class object in native code. If you invoke a native function, which is declared as a `static native` method in a Java class, your native function always gets the reference of the class object as the second parameter. Sometimes, you may have a reference of a Java object in the `jobject` type and you may want to gets its class object reference. You need to use the `GetObjectClass()` JNI function to get the reference of the class object of a Java object as shown below.

```
jobject obj = get the reference to a Java object...;
jclass cls = env->GetObjectClass(obj);
```

Sometimes, you may have to get the reference of a class object using the class name. In that case, you need to use the `FindClass()` JNI function. You need to use the fully qualified name of the class in the `FindClass()` method by replacing a dot in the package name by a forward slash. If you are trying to get the reference of a class object for an array, you need to use the array class signature. For example, to get the reference of the class object for the `java.lang.String` class, you need to use `"java/lang/String"` as the class name. To get the class object reference for `int[]` you need to use `"[I"` as the class name. To know the correct signature for the class of an array type, you can declare a field in a class of that array type and use the `javap` command with the `-s` and `-private` options to get its signature. The following snippet of code demonstrates how to get the reference of the class object for some Java reference types.

```
jclass cls;

// Get the reference of the java.lang.String class object
cls = env->FindClass("java/lang/String");
```

```
// Get the reference of the int[] array class object
cls = env->FindClass("[I");

// Get the reference of the int[][] array class object
cls = env->FindClass("[[I");

// Get the reference of the String[] array class object.
// Note a semi-colon in signature
cls = env->FindClass("[Ljava/lang/String;");
```

Accessing Fields and Methods of the Java Object/Class

Before you can access the fields of a Java object/class in native code, you must get the field ID. You need to use the `GetFieldID()` JNI function to get the field ID of an instance field and the `GetStaticFieldID()` JNI function to get the field ID for a `static` field. Signatures of these two methods are as follows:

```
jfieldID GetFieldID(jclass cls, const char *name, const char *sig)
jfieldID GetStaticFieldID(jclass cls, const char *name, const char *sig)
```

The `cls` parameter is the reference of the class object, which defines the instance/`static` field. The `name` parameter is the name of the field. The `sig` parameter is the signature of the field. You need to use the `javap` command with the `-s` and `-private` options to get the signature of a field defined in a class.

You need to use a `Get<XXX>Field()` JNI function to get the value of an instance field and a `GetStatic<XXX>Field()` JNI function to get the value of a `static` field, where `<XXX>` is the type of field whose value could be - `Boolean`, `Byte`, `Char`, `Double`, `Float`, `Int`, `Long`, `Short`, or `Object`. The `Set<XXX>Field()` and `SetStatic<XXX>Field()` JNI functions let you set the value of instance and `static` fields respectvely. The signature for all these four methods are as follows. Here `<RRR>` is a native data type, for example, if `<XXX>` is `int`, `<RRR>` is `jint`.

```
<RRR> Get<XXX>Field(jobject obj, jfieldID fieldID)
<RRR> GetStatic<XXX>Field(jclass clazz, jfieldID fieldID)
void Set<XXX>Field(jobject obj, jfieldID fieldID, <RRR> value)
void SetStatic<XXX>Field(jclass clazz, jfieldID fieldID, <RRR> value)
```

Suppose `obj` is an instance of `jobject` (that is, a Java object reference) and `cls` is its class reference. There are two fields, `num` and `count`, of type `int` in the class represented by `cls`. The `num` field is an instance field and `count` field is a `static` field. The following snippet of code shows how to access these two fields in native code and increment their values by 1.

```
// Get the field ID of num and count fields
jfieldID numFieldId = env->GetFieldID(cls, "num", "I");
jfieldID countFieldId = env->GetStaticFieldID(cls, "count", "I");

// Get the field values
jint numValue = env->GetIntField(obj, numFieldId);
jint countValue = env->GetStaticIntField(cls, countFieldId);

// Increment the values by 1 and set them back to the fields
numValue = numValue + 1;
countValue = countValue + 1;
```

```
env->SetIntField(obj, numFieldId, numValue);
env->SetStaticIntField(cls, countFieldId, countValue);
```

The steps to use a method of Java object/class in native code are similar to using their fields. You need to get the method ID of a method before you can access the method. You can use `GetMethodID()` and `GetStaticMethodID()` JNI functions to get the method ID for an instance method and a `static` method, respectively. Their signatures are as follows:

```
jmethodID GetMethodID(jclass clazz, const char *name, const char *sig)
jmethodID GetStaticMethodID(jclass clazz, const char *name,
                            const char *sig)
```

The `name` of the method is its simple name and its signature can be obtained using the `javap` command with the `-s` and `-private` options. The following snippet of code shows how to get the method ID from a few methods of a Java class assuming that `cls` represents the class object reference.

```
jmethodID methodID

// Method is "void objectCallBack()"
methodID = env->GetMethodID(cls, "objectCallBack", "()V");

// Method is "static void classCallBack()"
methodID = env->GetStaticMethodID(cls, "classCallBack", "()V");

// Method is "int getLength(String str)"
methodID = env->GetMethodID(cls, "getLength", "(Ljava/lang/String;)I");

// Method is "int[] increment(int[], int)"
methodID = env->GetMethodID(cls, "increment", "([II)[I");
```

Calling an instance or `static` method is easy. You need to use an object/class, the method ID and method arguments, if any, to call a method. You can use any of the following methods to call an instance method of an object.

```
<RRR> Call<XXX>Method(jobject obj, jmethodID methodID, ...);
<RRR> Call<XXX>MethodA(jobject obj, jmethodID methodID, jvalue *args);
<RRR> Call<XXX>MethodV(jobject obj, jmethodID methodID, va_list args);
```

The `<XXX>` in the method name is the return type of the method and it could be - `Boolean`, `Byte`, `Char`, `Double`, `Float`, `Int`, `Long`, `Short`, `Object`, or `Void`. The `<RRR>` is the return type of the method and it could be `jboolean`, `jbyte`, `jchar`, `jdouble`, `jfloat`, `jint`, `jlong`, `jshort`, `jobject`, or `void` depending on the corresponding `<XXX>` value. The difference between `Call<XXX>Method()`, `Call<XXX>MethodA()` and `Call<XXX>MethodV()` is how you want to pass the parameters to the method. The `Call<XXX>Method()` method lets you pass parameters to a method as a comma separated list. The `Call<XXX>MethodA()` method lets you pass parameters to a method as an array of `jvalue` type. The `Call<XXX>MethodV()` method lets you pass parameters to a method as `va_list`. The following snippet of code shows how to call an instance method assuming that `obj` is a reference of `jobject` type and the method ID is `methodID`.

```
// Method is "void m1()"
env->CallVoidMethod(obj, methodID);
```

```
// Method is "void m2(int a)"
env->CallVoidMethod(obj, methodID, 109);

// Method is "int m2(double a)"
jint value = env->CallIntMethod(obj, methodID, 109.23);
```

Calling a `static` method is similar to calling an instance method. You need to use a class object reference to call a `static` method. You need to use one of the following JNI functions to call a `static` method. Note that the JNI function names, which are used to call `static` methods, contain the word "`Static`".

```
<RRR> CallStatic<XXX>Method(jclass cls, jmethodID methodID, ...);
<RRR> CallStatic<XXX>MethodA(jclass cls, jmethodID methodID,
                             jvalue *args);
<RRR> CallStatic<XXX>MethodV(jclass cls, jmethodID methodID,
                             va_list args);
```

JNI lets you call an instance method on an object from any class in its class hierarchy. When you use a `Call<XXX>Method()` function, it uses the object's class to call the method. Consider the following class hierarchy.

```
//A.java
package com.jdojo.chapter6;

public class A {
     public int m1() {
          return 1;
     }
}

//B.java
package com.jdojo.chapter6;

public class B extends A {
     public int m1() {
          return 3;
     }
}

//C.java
package com.jdojo.chapter6;

public class C extends B {
     public int m1() {
          return 3;
     }
}
```

Class `B` and `C` override the `m1()` method. If you use `CallIntMethod()` to call the `m1()` method of an object of class `C`, it will call `m1()` method in class `C` and it returns 3. JNI lets you call `m1()` method in class `A` or class `B` using an object of class `C`. To call a method on an object from its superclass, you need to use one of the following JNI methods.

```
<RRR> CallNonvirtual<XXX>Method(jobject obj, jclass cls,
                                jmethodID methodID, ...)
```

```
<RRR> CallNonvirtual<XXX>MethodA(jobject obj, jclass cls,
                                 jmethodID methodID, jvalue *args)
<RRR> CallNonvirtual<XXX>MethodV(jobject obj, jclass cls,
                                 jmethodID methodID, va_list args)
```

You need to use the reference of the object and its class in these versions of the methods. The `methodID` must be obtained using the class from which the method needs to be called. For example, the following code calls the `m1()` method from class `B` on an object of class `C`. The following sample code also creates an object of class `C`.

```
// Get the class references for B and C
jclass bCls = env->FindClass("com/jdojo/chapter6/B");
jclass cCls = env->FindClass("com/jdojo/chapter6/C");

// Get method ID for the constructor of class C
jmethodID cConstrctorID = env->GetMethodID(cCls, "<init>", "()V");

// Create an object of class C
jobject cObject = env->NewObject(cCls, cConstrctorID);

// Get the method ID for the m1() method in class B
jmethodID bMethodID = env->GetMethodID(bCls, "m1", "()I");

// Call the m1() method in class B using on object of class C
jint h = env->CallNonvirtualIntMethod(cObject, bCls, bMethodID);

// will print 2, which is returned from m1() in class B
printf("%i\n", h);
```

Let us look at a complete example of accessing fields and methods of a Java object in native code. Listing 6-7 contains the Java code, which has two fields and two methods – `num, count, objectCallBack()` and `classCallBack()`, which will be accessed from native code. It has a native method called `callBack()`. The `callBack() native` method increments the `num` and `count` fields by 1 and calls the `objectCallBack() and classCallBack()` methods.

Listing 6-7: Accessing fields and methods of Java objects/classes from native code

```
// JNIJavaObjectAccessTest.java
package com.jdojo.chapter6;

public class JNIJavaObjectAccessTest {
     static {
          System.loadLibrary("jniutility");
     }

     private int num = 10;
     private static int count = 1001;

     public void objectCallBack() {
          System.out.println("Inside objectCallBack() method.");
     }

     public static void classCallBack() {
          System.out.println("Inside classCallBack() method.");
     }
```

```
    public native void callBack();

    public int hashCode() {
        return -9999;
    }

    public static void main(String[] args) {
        JNIJavaObjectAccessTest test = new JNIJavaObjectAccessTest();

        System.out.println("Before calling native method...");
        System.out.println("num = " + test.num);
        System.out.println("count = " + count);

        // Call native method
        test.callBack();

        System.out.println("After calling native method...");
        System.out.println("num = " + test.num);
        System.out.println("count = " + count);
    }
}
```

```
Output:

Before calling native method...
num = 10
count = 1001
Inside objectCallBack() method.
Inside classCallBack() method.
After calling native method...
num = 11
count = 1002
```

Listing 6-8: C++ implementation of the callBack() native method declared in JNIJavaObjectAccessTest class

```
JNIEXPORT void JNICALL
Java_com_jdojo_chapter6_JNIJavaObjectAccessTest_callBack
(JNIEnv *env, jobject obj) {
    jclass cls;

    // Get the class reference for the object
    cls = env->GetObjectClass(obj);
    if ( cls == NULL) {
        return;
    }

    // Access the fields
    jfieldID numFieldId = env->GetFieldID(cls, "num", "I");
    jfieldID countFieldId = env->GetStaticFieldID(cls, "count", "I");

    jint numValue = env->GetIntField(obj, numFieldId);
    jint countValue = env->GetStaticIntField(cls, countFieldId);

    numValue = numValue + 1;
    countValue = countValue + 1;
```

```
    env->SetIntField(obj, numFieldId, numValue);
    env->SetStaticIntField(cls, countFieldId, countValue);

    // Call the instance method
    jmethodID instanceMethodID = env->GetMethodID(cls,
                                                  "objectCallBack",
                                                  "()V");
    if(instanceMethodID != 0) {
        env->CallVoidMethod(obj, instanceMethodID);
    }

    // Call the static method
    jmethodID staticMethodID = env->GetStaticMethodID(cls,
                                                      "classCallBack",
                                                      "()V");
    if(staticMethodID != 0) {
        env->CallStaticVoidMethod(cls, staticMethodID);
    }

    return;
}
```

Creating Java Objects in Native Code

JNI lets you create Java objects in native code. You create an object without invoking any constructor or by invoking a specific constructor. You need to use the `AllocObject()` JNI function to allocate memory for a Java object without invoking any of its constructor. Note that all instance fields will have their default values according to their data types. Instance fields would not be initialized when you use `AllocObject()` JNI function and no instance initializer will be invoked either. Here is the snippet of code to allocate memory for an object of a class in Java:

```
jclass cls = get the class reference...;
jobject obj = env->AllocObject(cls);
if (obj == NULL) {
    // Object could not be created. Handle the error condition
}
```

You can create a Java object by invoking a specific constructor of a Java class using one of the following JNI functions. The following functions to create a Java object differ only in how to pass the parameters for a constructor.

```
jobject NewObject(jclass clazz, jmethodID methodID, ...);
jobject NewObjectA(jclass clazz, jmethodID methodID, jvalue *args);
jobject NewObjectV(jclass clazz, jmethodID methodID, va_list args);
```

The `methodID` parameter is the method ID for the constructor that you want to invoke as part of creating the new object. There is a special string that is used for a method name when you want to get the method ID for a constructor of a class. You need to use "`<init>`" or "`$init$`" as the method name for a constructor. Consider the code for a class named `ABC` as shown in Listing 6-9.

Listing 6-9: A sample class to demonstrate the Java object creation in native code

```
// ABC.java
package com.jdojo.chapter6;
```

```
public class ABC {
     private int value = -1;

     public ABC() {
     }

     public ABC(int value) {
          this.value = value;
     }

     public int getValue() {
          return value;
     }
}
```

You can get the class reference of `ABC` in native C++ code as shown below.

```
jclass abcCls = env->FindClass("com/jdojo/chapter6/ABC");
```

The following C++ code allocates memory for an `ABC` object without invoking a constructor.

```
jobject abcObject = env->AllocObject(abcCls);
```

At this point, `abcObject` exists in memory and its instance field, `value`, still has its default value of 0. If you call the `getValue()` method on `abcObject` at this point, it will return 0 and not –1 as you might expect.

You need to use the `NewObject()` JNI function if you want to create an object of a Java class by invoking one of its constructors. The following snippet of code creates an object of the `ABC` class by invoking its no-args constructor. The method name for the constructor is always either `"<init>"` or `"$init$"`. The signature for a constructor depends on the number and type of parameters it accepts. For the no-args constructor, the signature is `"()V"`. If a constructor accepts one `int` parameter, its signature would be `"(I)V"`. You can get the signature of a constructor of a class by using the `javap` command with the `-s` option. Use the `-private` option with `javap` if you also want to include the `private` member's signatures.

```
// Get the method ID for the default constructor of class ABC
jmethodID mid = env->GetMethodID(abcCls, "<init>", "()V");

// Create an object of class ABC using the default constructor
jobject abcObject = env->NewObject(abcCls, mid);
```

At this point, if you call the `getValue()` method on `abcObject`, it will return -1, which is the initial value of the `value` instance field. When a constructor is called, all instance fields are initialized.

The following snippet of code calls the second version of the constructor of the `ABC` class, which accepts an `int` parameter. It passes 999 as the value for the parameter for the `"ABC(int value)"` constructor.

```
// Get the method ID for the constructor for class ABC
jmethodID abcConstrctorID = env->GetMethodID(abcCls, "<init>","(I)V");

// Create an object of class ABC passing 999 to the constructor
```

```
jobject abcObject = env->NewObject(abcCls, abcConstrctorID, 999);
```

At this point, if you call the `getValue()` method on `abcObject`, it will return 999, which is set in its constructor during its creation.

TIP
You need to use the `NewObjectArray()` JNI function to create an array of a specific type. The `AllocObject()` and `NewObject()` JNI functions can be used only to create objects of a non-array reference type.

Exception Handling in Native Code

JNI lets you handle exceptions in native code. Native code can detect and handle exceptions that are thrown in the JVM as a result of calling a JNI function. Native code can also throw an exception that can be propagated to Java code. Exception handling mechanism in the native code differs from that of the Java code. When an exception is thrown in Java code, the control is transferred immediately to the nearest `catch` block that can handle the exception. When an exception is thrown during native code execution, the native code keeps executing and the exception remains pending until the control returns to the Java code. Once an exception is pending, you should not execute any other JNI functions except the ones that free native resources. There are two ways to detect if an exception has occurred as a result of a JNI function call in the native code.

- Some JNI functions return a special value if an exception occurs. For example, if you call the `FindClass()` JNI function and the class is not found, any one of the four exceptions may be thrown - `ClassFormatError, ClassCircularityError, NoClassDefFoundError, or OutOfMemoryError`. However, the `FindClass()` JNI function returns `NULL` as a special value, if any of the four exceptions is thrown. You should check for `NULL` as a return value just after a call to the `FindClass()` JNI function and write code to handle the exception. Typically, you return the control to the caller, so that the caller can handle the exception as shown below:

  ```
  jclass cls = env->FindClass("abc/xyz/NonExistentClass");
  if (cls == NULL) {
     // Here, free up any resources you had held and return.
     // Exception is pending at this time. It will be thrown
     // when the control returns to the Java code.
     return;
  }
  ```

- In some cases, it is not possible to return a special value from a JNI function to indicate that an exception has occurred. Suppose you are accessing a Java array in native code and you have exceeded the array's boundary. In this case, an exception of type `ArrayIndexOutOfBoundsException` is thrown by the JVM. You may call a method of a Java object where an exception occurs. In such cases, you need to use either `ExceptionOccurred()` or `ExceptionCheck()` JNI function immediately after such JNI function call to check if an exception has occurred. These functions have the following signatures:

  ```
  jthrowable ExceptionOccurred()
  jboolean ExceptionCheck()
  ```

  If an exception has occurred, the `ExceptionOccurred()` function returns the reference of

that exception object. Otherwise, it returns `NULL`. If an exception has occurred, the `ExceptionCheck()` function returns `JNI_TRUE`. Otherwise, it returns `JNI_FALSE`. The following snippet of code demonstrates how to use these two functions. You only need to use one of the two functions and not both at the same time.

```
/* Call a JNI function, which may throw an exception */

jthrowable e = env->ExceptionOccurred();
if (e != NULL) {
   // Free up any resources that you had held and return.
   // Exception is pending at this time. It will be thrown
   // when the control returns to the Java code.
   return;
}

/* Call a JNI function, which may throw an exception */

jboolean gotException = env->ExceptionCheck();
if (gotException) {
   // Free up any resources that you had held and return.
   // Exception is pending at this time. It will be thrown
   // when the control returns to the Java code.
  return;
}
```

Once you have detected an exception that has occurred in native code, you have three options:

- You can clear the exception and handle the exceptional condition in the native code. You can clear a pending exception using the `ExceptionClear()` JNI function as shown below. Once you clear the exception, that exception is not pending anymore.

  ```
  //Call a JNI function, which may throw an exception...
  jboolean gotException = env->ExceptionCheck();
  if (gotException) {
     // Clear the exception
     env->ExceptionClear();
     /* Write some code to take care of the exceptional condition */
  }
  ```

- You can return the control to the caller by using a `return` statement and let the caller handle the exception as follows:

  ```
  // Call a JNI function, which may throw an exception...
  jboolean gotException = env->ExceptionCheck();
  if (gotException) {
     // Free up any resources that you had held and return.
     // Exception is pending at this time. It will be thrown
     // when the control returns to the caller
     return;
  }
  ```

- You can handle the exception in the native code, clear the exception and throw a new exception. Note that throwing an exception from the native code does not transfer the control back to the Java code. You must write code (e.g. a `return` statement) to transfer the control back to the Java code, so that the exception you throw is handled in Java. You can throw an

exception in the native code using either of the following two JNI functions. Both functions return zero on success and a negative integer on failure.

```
jint Throw(jthrowable obj);
jint ThrowNew(jclass clazz, const char *message);
```

The `Throw()` function accepts a `jthrowable` object. The `ThrowNew()` function accepts the exception's class reference and a message. The following snippet of code shows how to throw a `java.lang.Exception` using the `ThrowNew()` function.

```
if (someErrorConditionIsTrue) {
   jclass cls = env->FindClass("java/lang/Exception");

   /* Check for exception here (omitted) */

   env->ThrowNew(cls, "your error message goes here");
   return;
}
```

TIP

If you want to print the stack trace of an exception from the native code, you can use the `ExceptionDescribe()` JNI function. It prints an exception stack trace on the standard error. If you want to raise a fatal error from the native code, you can use the `FatalError(const char *msg)` JNI function. The `FatalError()` function does not return and the JVM will not recover from this error either. A native method declared in Java code can also use a `throws` clause the same way as a Java non-native method can. The following is a valid native method declaration inside a Java class.

```
public native int myMethod() throws Exception;
```

Behind the Scenes

Creating an Instance of a JVM in Native Code

So far, we have seen Java applications using native code. Now, we are ready to see the reverse. That is, a native application using Java code. Why would you use Java code from a native application? You may want to use Java code from a native application for the following reasons.

- You may already have an application coded in Java and you want to use the existing code.
- Java provides a rich set of class libraries. You may want to take advantage of Java class libraries in your native application.

The part of JNI API that lets you create and load a JVM in native code is known as *Invocation API*. JNI lets you embed a JVM inside a native application. That is, you can create a JVM from a native application and use Java classes, as you use them in a Java application. It takes just a few lines of code to create a JVM in native code. All you need to do is to prepare the initial arguments that you want to pass to a JVM and call the `JNI_CreateJavaVM()` Invocation API function to create the JVM.

The initial argument that is passed to a JVM is a `JavaVMInitArgs` structure, which is defined as follows.

```
typedef struct JavaVMInitArgs {
     jint version;
     jint nOptions;
     JavaVMOption *options;
     jboolean ignoreUnrecognized;
} JavaVMInitArgs;
```

The version field of the `JavaVMInitArgs` structure indicates the JNI version and it must be set to at least JNI_VERSION_1_2. The `nOptions` field is set to the number of options you want to pass to a JVM. The options field is an array of a `JavaVMOption` structure, which is defined as follows:

```
typedef struct JavaVMOption {
     char *optionString;
     void *extraInfo;
} JavaVMOption;
```

If `ignoreUnrecognized` is set to `JNI_TRUE`, the `JNI_CreateJavaVM()` function will ignore the unrecognized options. If it is set to `JNI_FALSE`, the `JNI_CreateJavaVM()` function will return `JNI_ERR` as soon as it encounters an unrecognized option.

The `optionString` field in the `JavaVMOption` structure is a string that is the value for the option to a JVM in the default platform encoding.

The `extraInfo` field is used for special kinds of JVM arguments. It represents a function hook for redirecting a JVM message, a JVM exit hook, or a JVM abort hook. The type of hook the `extraInfo` field represents depends on the value for the `optionString` field. If the `optionString` field has value of "`vfprintf`", "`exit`", or "`abort`", the `extraInfo` field represents a JVM message redirection hook, JVM exit hook or JVM abort hook respectively. Note that `vfprintf` hook redirects only the JVM message to the hook. It does not redirect the `System.out` and `System.err` messages to the hook. If you have set a `vsprintf` hook in native code and if you have used one of `print()/println()` methods of `System.out/System.err` in Java code, those messages would not be redirected to your `vfprintf` hook. You need to use the `setOut()` and `setErr()` methods of the `System` class to redirect `System.out` and `System.err` messages. The exit hook for a JVM is called upon a normal termination of the JVM such as by calling the `System.exit(int exitCode)` method in Java code. The abort hook for a JVM is called upon abnormal termination of the JVM. The following snippet of code shows how to populate the `extraInfo` field with a different VM hook. First, we define three functions that will serve as the three types of hooks. Note that the functions must have the same signatures as shown in the following snippet of code.

```
jint JNICALL jvmMsgRedirection_hook(FILE *stream, const char *format,
                                    va_list args) {
     /* You can log the VM message here...*/
     // We will just print the VM message on the standard output
     return vfprintf(stdout, format, args);
}

void JNICALL jvmExit_hook(jint code) {
     // You can do some cleanup work here...

     printf("VM exited with exit code %i\n", code);
```

```
}

void JNICALL jvmAbort_hook() {
     printf("VM was aborted\n");
}
JavaVMOption jvmOption[3];

// Add JVM hooks
options[0].optionString = "vfprintf";
options[0].extraInfo    = jvmMsgRedirection_hook;

options[1].optionString = "exit";
options[1].extraInfo    = jvmExit_hook;

options[2].optionString = "abort";
options[2].extraInfo    = jvmAbort_hook;
```

The following snippet of code shows how to populate a `JavaVMInitArgs` structure with initial arguments for a JVM. It sets only two arguments `java.class.path` and `java.lib.path`. You can set more JVM arguments if you need to.

```
// Populate the JVM options in JavaVMOption structure
const jnit MAX_OPTIONS = 2; // will pass two arguments to the JVM

JavaVMOption options[MAX_OPTIONS];

// Our first argument is java.class.path (CLASSPATH for JVM)
options[0].optionString = "-Djava.class.path=.;c:\\myjni\classes";

// Our second argument is java.library.path
// (PATH to find a shared library)
options[1].optionString = "-Djava.library.path=c:\\myjni\\libs";

// Populate JavaVMInitArgs structure with options details
JavaVMInitArgs vm_args;
vm_args.version  = JNI_VERSION_1_2;
vm_args.nOptions = MAX_OPTIONS;
vm_args.options  = options;
vm_args.ignoreUnrecognized = true;
```

Once you have the JVM arguments ready in a `JavaVMInitArgs` structure, you are just one JNI function call away from creating a JVM in your native code. The `JNI_CreateJavaVM()` JNI function accepts three parameters. The first parameter is a pointer to a `JavaVM` structure that represent the JVM. The second parameter is a pointer to a `JNIEnv` structure, which is the JNI interface. The third parameter is the initial argument to the JVM. The following snippet of code shows how to create a JVM in native code. You need to check for any errors that the `JNI_CreateJavaVM()` function might return. It returns `JNI_ERR` if cannot create a JVM.

```
JNIEnv *env;
JavaVM *jvm;
long status;
status = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);

if (status == JNI_ERR ) {
  printf("Could not create VM. Exiting application...\n");
```

```
  return 1;
}
```

Once you get the `JNIEnv` structure, you can use it to find a class, create an object of that class and execute any methods on that object. In fact, it lets you access a JVM using JNI.

After you are done with the JVM, you need to destroy it:

```
// Destroy JVM
jvm->DestroyJavaVM();
```

Listing 6-10 has a Java class with a `printMsg() static` method to print a message on the standard output. The C++ console application as listed in Listing 6-11 creates a JVM and calls the `printMsg()` method of the `EmbeddedJVMJNI` class. The code listed in Listing 6-11 has been tested using Visual C++ 2005 on a Windows platform. When you compile this program, you would need to provide the path for the `jvm.lib` file, which is installed in `JAVA_HOME\lib` directory. When you run the application, it will look for the `jvm.dll` shared library, which is found in `JRE_HOME\bin\client` directory for Java 6. If you are using a diferent version of Java, you may find it in other subdirectories of `JRE_HOME` directory. You need to include the directory that contains the `jvm.dll` file in the `PATH` environment variable when you run this program. If you run and compile this program on a platform other than Windows, you need to change the path separator in the `CLASSPATH` value. The path separator is a semi-colon (`;`) on Windows and a colon (`:`) on UNIX.

Listing 6-10: A Java class, whose method will be called from a native application using an embedded JVM

```
// EmbeddedJVMJNI.java
package com.jdojo.chapter6;

public class EmbeddedJVMJNI {
     public static void printMsg(String msg) {
          System.out.println(msg);
     }
}
```

Listing 6-11: Embedding JVM in a native application.

```
#include <jni.h>
#pragma comment(lib, "jvm.lib")

int main(int argc, char **argv) {
     // We will pass two arguments to the JVM
     const jint MAX_OPTIONS = 1 ;
     JavaVMOption options[MAX_OPTIONS];
     options[0].optionString = "-Djava.class.path=.;c:\\myclasses";

     // Prepare the JVM initial arguments
     JavaVMInitArgs vm_args;
     vm_args.version  = JNI_VERSION_1_2;
     vm_args.nOptions = MAX_OPTIONS;
     vm_args.options  = options;
     vm_args.ignoreUnrecognized = true;

     // Create the JVM
     JavaVM *jvm;
```

```
    JNIEnv *env;
    long status = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
    if (status == JNI_ERR) {
        printf("Could not create VM. Exiting application...\n");
        return 1;
    }

    const char *className = "com/jdojo/chapter6/EmbeddedJVMJNI";
    jclass cls = env->FindClass(className);
    if (cls == NULL) {
        // Print exception stack trace and destroy the JVM
        env->ExceptionDescribe();
        jvm->DestroyJavaVM();
        return 1;
    }

    if (cls != NULL) {
        jmethodID mid = env->GetStaticMethodID(cls, "printMsg",
                                    "(Ljava/lang/String;)V");
        if(mid != NULL) {
            jstring m = env->NewStringUTF("Hello from C++...\n");
            env->CallStaticVoidMethod(cls, mid, m);
            if (env->ExceptionCheck()) {
                env->ExceptionDescribe();
                env->ExceptionClear();
            }
        }
    }

    // Destroy JVM
    jvm->DestroyJavaVM();
    return 0;
}
```

Synchronization in Native Code

JNI provides two functions – `MonitorEnter()` and `MonitorExit()`, to synchronize access to native code in a multi-threaded environment. These functions are used in tandem and their use is equivalent to using the `synchronized` keyword in Java code. The signatures of these two functions are as follows:

```
jint MonitorEnter(jobject obj);
jint MonitorExit(jobject obj);
```

Both functions return 0 (`JNI_OK` is defined as 0 in the `jni.h` header file) on success and a negative number on failure. You must check their return values to handle the code synchronization properly. Here is the sample Java code that uses synchronization.

```
Object someObject = get the reference of a java object;
/* Other logic goes here...*/
synchronized(someObject) {
    //Synchronized code goes here...
}
```

The equivalent native code is as follows.

```
jobject someObject = get the reference of a java object;
/* Other logic goes here...*/
jint enterStatus = env->MonitorEnter(someObject);
if (enterStatus != JNI_OK) {
     // Handle the error condition here...
}

// Synchronized codes go here...

jint exitStatus = env->MonitorExit(someObject);
if (exitStatus != JNI_OK ) {
     // Handle the error condition here...
}
```

There are no equivalent JNI functions for Java `wait()` and `notify()` to aid in thread synchronization. However, you can always invoke these two Java methods from native code.

References

Berners-Lee, et al. *Uniform Resource Identifier (URI).* http://www.ietf.org/rfc/rfc3986.txt (accessed September 2011).

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

A

`Accept` 21
Accessing Java Objects in Native Code 120
Accessing the Contents of a URL 51
address mask 13
AfriNIC 11
Anycast IP Address 17
APNIC 11
ARIN 11
Asynchronous Client Socket Channel 80
Asynchronous Socket Channels 72
Asynchronous Transfer Mode 7

B

`Bind` 21
broadcast 14
Broadcast IP Address 17
bus network 7

C

C/C++ 97
Campus Area Network 7
CIDR (Classless Inter-Domain Routing) 14
classful IP address 12
Classless Inter-Domain Routing (CIDR) 14
`Close` 21
`Connect` 21
Connected UDP Socket 41
connection socket 27
connectionless protocol 34
connectionless socket 19, 34
connection-oriented socket 19
Creating a Shared Library 104
Creating C/C++ Header File 102
Creating JVM in Native Code 130

D

datagram 10
Datagram Channels
  multicasting 91
datagram socket 19
Datagram-Oriented Socket Channel 85
`DatagramSocket` class 34
Domain Name System (DNS). 23
Dynamic and/or Private Ports 18
dynamic link library (`DLL`) 97

E

Ethernet 7
Exception Handling in Native Code 128

F

Fiber Distributed Data Interface 7
File Transfer Protocol 9
frame 10

G

gateways 7
Getting Started with JNI 98
Gopher 9

H

host 7
hybrid network 7
Hypertext Transfer Protocol 9

I

IANA 11
ICMP (Control Message Protocol) 9
IGMP (Internet Group Management Protocol) 9
`InetAddress` class 23
`InetSocketAddress` 25
internet 7
Internet 7
Internet Assigned Numbers Authority 11
Internet Control Message Protocol 9
Internet Group Management Protocol 9
Internet Protocol – IPv4 11
Internet Protocol next generation 11
Internet Protocol Security 9
Internet Protocol Suite 8
Internet Reference Model 8
Internet Stream Protocol 11
internetwork 7
internetworking 7
Invocation API 130
IP address 10, 11
IP Addressing Scheme 11
IP layer 10
IPng 11
IPng (Internet Protocol next generation) 11
IPsec (Internet Protocol Security) 9
IPv4 11
IPv4 Addressing Scheme 12

IPv6 11
IPv6 and its Addressing Scheme 14

J

`java.library.path property` 99
JavaVMInitArgs structure 131
`jboolean type` 112
`jbyte type` 112
`jchar` type 112
`jclass` 103
`jdouble` type 112
`jfloat type` 112
`jint type` 112
`jlong` type 112
JNI 97
  Accessing Fields and Methods of Java Object/Class 121
  Accessing Java Objects in Native Code 120
  Creating a Shared Library 104
  Creating C/C++ Header File 102
  Creating Java Objects in native code 126
  Creating JVM in Native Code 130
  Data Types Mapping 111
  Exception Handling in Native Code 128
  Getting Class Reference 120
  Getting Started 98
  Running Java Program 108
  Synchronization in Native Code 134
  System Requirements 98
  Using JNI Functions in C/C++ 113
  Working with Arrays 116
  Working with Strings 113
  Writing Java Program 98
  Writing the C/C++ Program 103
JNI (Java Native Interface) 97
JNI architecture 97
`JNI_CreateJavaVM()` 130
JNICALL 103
JNIEXPORT 103
`jobject` 103
`jshort` type 112
JSP 53
`jstring type` 112
`jthrowable type` 112

L

LACNIC 11
`Listen` 21
`load()` method 99
`loadLibrary()` method 98
Local area network 7
LocalTalk 7
Loopback IP Address 15

M

MAC (Media Access Address) 10
Media Access Address 10
Metropolitan Area Network 7
multicast 14
multicast group 42
Multicast IP Address 16
Multicast Socket
  UDP 42
Multicasting using Datagram Channels 91

N

JNI 109
Native Function Naming Rules 109
`native` keyword 99
`native` method 99
network 7
network classes 12
Network News Transfer Protocol 9
Network Programming 7
`NewObject()` JNI method 127
Non-Blocking Socket Programming 59

P

Packet 10
packet switching networks 8
port 18
Port Numbers 18
protocol 8
protocol suite 8

R

`Receive/ReceiveFrom` 21
Regional Internet Registry 11
Registered Ports 18
Representing a Computer Address 23
Representing a Socket Address 25
ring network 7
RIPE NCC 11
RIR 11
routers 7

S

SCTP (Stream Control Transmission Protocol) 9
segment 10
`SelectionKey` 60
`Selector` 60
self-identifying IP Address 12
`Send/Sendto` 21
server socket 23
`ServerSocketChannel` 60
shared object (SO) 97
Simple Mail Transfer Protocol 9

SMTP 9
`Socket` 21
Socket API 19
  Client-Server Paradigm 19
Socket Security Permissions 71
`SocketChannel` 60
Special IP Addresses 15
stack of protocols 8
star network 7
Stream Control Transmission Protocol 9
stream socket 19
subnet mask 13
subnets 13
subnetting 13
supernetting 13, 14
Synchronization in Native Code 134

T

TCP Client Socket 31
TCP Server Socket 26
TCP/IP Layering Model 8
Telecommunication Network 9
Token Ring 7
tree network 7
TTL (Time To LIve) 9

U

UDP (User Datagram Protocol) 9
UDP Echo Server 37
UDP Multicast Socket 42
UDP Sockets 34
  connected 41
Unicast IP Address 16
Uniform Resource Identifier (URI) 44
Uniform Resource Locator (`URL`) 44
Uniform Resource Name (URN) 45
UNIX 97
Unspecified IP Address 18
URI 44
`URI class` 48
URL 44
  accessing contents 51
`URL class` 48
`URLConnection` object 57
`URLDecoder class` 48
`URLEncoder class` 48
URN 44
User Datagram Protocol 9
Using JNI Functions in C/C++ 113

V

virtual connection 20
Visual Studio 2005 104

W

Well-known Ports 18
Wide Area Network 7
Win32 97
Writing the C/C++ Program 103