Complete reference java

Science, Computer Science

Java 2: The Complete Reference by Patrick Naughton and Herbert Schildt Osborne/McGraw-Hill © 1999, 1108 pages ISBN: 0072119764 This thorough reference reads like a helpful friend. Includes servlets, Swing, and more. Table of Contents Back Cover Synopsis by Rebecca Rohan Java 2: The Complete Reference blends the expertise found in Java 1: The Complete Reference with Java 2 topics such as " servlets" and " Swing. " As before, there's help with Java Beans and migrating from C++ to Java. A special chapter gives networking basics and breaks out networking-related classes.

This book helps you master techniques by doing as well as reading. Projects include a multi-player word game with attention paid to network security. The book is updated where appropriate throughout, and the rhythm of text, code, tables, and illustrations is superb. It's a valuable resource for the developer who is elbow-deep in demanding projects. Table of Contents Java 2 Preface - 7 Part l The Java Language - The Complete Reference - 4 Chapter 1 Chapter 2 Chapter 3 Chapter 4 Chapter 5 Chapter 6 Chapter 7 Chapter 8 Chapter 9 hapter 10 The Genesis of Java - 9 - An Overview of Java - 20 - Data Types, Variables, and Arrays - 36 - Operators - 57 - Control Statements - 75 - Introducing Classes - 94 - A Closer Look at Methods and Classes - 111 - Inheritance - 134 - Packages and Interfaces - 156 - Exception Handling - 174 Chapter 11 - Multithreaded Programming - 188 Chapter 12 - I/O, Applets, and Other Topics - 214 Part ll The Java Library Chapter 13 - String Handling - 235 Chapter 14 - Exploring java. lang - 255 Chapter 15 - java. util Part 1: The Collections Framework - 297 Chapter 16 - java. util Part 2: More Utility Classes - 343 2- Chapter 17 - Input/Output: Exploring java. io - 362 Chapter 18 - Networking - 397 Chapter 19 - The Applet Class - 426 Chapter 20 - Event

Handling - 443 Chapter 21 - Introducing the AWT: Working with Windows, Graphics, and Text - 466 Chapter 22 - Using AWT Controls, Layout Managers, and Menus - 499 Chapter 23 - Images - 543 Chapter 24 - Additional Packages - 568 Part lll Software Development Using Java Chapter 25 - Java Beans - 582 Chapter 26 - A Tour of Swing - 601 Chapter 27 - Servlets - 616 Chapter 28 - Migrating from C++ to Java - 641 Part lV Applying Java

Chapter 29 - The DynamicBillboard Applet - 659 Chapter 30 - ImageMenu: An Image-Based Web Menu - 683 Chapter 31 - The Lavatron Applet: A Sports Arena Display - 689 Chapter 32 - Scrabblet: A Multiplayer Word Game - 696 Appendix A - Using Java’s Documentation Comments - 739 Back Cover Master Java with the most comprehensive all-in-one tutorial/reference available, now completely updated for the Java 2 specification. Top programming experts Patrick Naughton and Herbert Schildt show you everything you need to know to develop, compile, debug and run Java applications and applets.

Inside you'll find a complete description of the Java language, its class libraries, and its developmentenvironment. With clear descriptions, hundreds of practical examples, and expert techniques, this is a book that no Java programmer should be without. With this book, you'll: • • • • • • • Master the Java language and its core libraries Create portable Java applets and applications Fully utilize the Abstract Window Toolkit (AWT) Supercharge your programs using multiple threads Effectively pply Java's networking classes Create servlets, draw images, and develop Java Beans Migrate code from C++ to Java Plus, you'll find details on new Java 2 features, including: •

• • • The powerful collections framework The Swing component set The Java threading model The numerous methods, classes, and interfaces found throughout the API About the Authors Patrick Naughton is currently the chieftechnologyoffice for Infoseek Corporation. He is the founding member of the original Sun Microsystems -3- project team that developed Java.

Herbert Schildt is a leading authority on C and C++, an expert on Windows, and master at Java. He has written numerous best-selling books. Java 2: The Complete Reference Third Edition Patrick Naughton Herbert Schildt Osborne/McGraw-Hill 2600 Tenth Street Berkeley, California 94710 U. S. A. For information on translations or book distributors outside the U. S. A. , or to arrange bulk purchase discounts for sales promotions, premiums, or fund-raisers, please contact Osborne/McGraw-Hill at the above address.

234567890 AGM AGM 90198765432109 ISBN 0-07-211976-4 Publisher Brandon A. Nordin Associate Publisher/Editor-in-Chief Scott Rogers Acquisitions Editor Megg Bonar Project Editor Janet Walden Editorial Assistant Stephane Thomas Technical Editor Tom Feng -4- Copy Editor William McManus Proofreader Emily K. Wolman Indexer Sheryl Schildt Computer

Designer Michelle Galicia Jani Beckwith Ann Sellers Illustrator Brian Wells Beth Young Information has been obtained by Osborne/McGraw-Hill from sources believed to be reliable.

However, because of the possibility of human or mechanical error by our sources, Osborne/McGraw-Hill, or others, Osborne/McGraw-Hill does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from use of such information. About the Authors Patrick Naughton started consulting as a software engineer in 1982, paying his way through school and gaining a trial-by-fire perspective on the PC industry as it grew from its infancy.

After extensive experience with the X Window System from MIT, he joined Sun Microsystems' window systems group in 1988. In late 1990, Naughton started a secret project called " Green" in SunLabs. This small project intended to create a completely new platform for software development that would solve many of the problems in existing systems. The most significant technology to come out of the Green project was Java. Naughton was instrumental in the creation and evolution of Java, from its inception through to its revolutionary transition into the language of the Internet.

Naughton is currently the executive vice president of products for Infoseek Corporation, and is the leader of the team that creates the groundbreaking GO Network™. Prior to working with Infoseek, he was president and chief technology officer of Paul Allen's Starwave Corporation, where he led the development of platform strategies, systems software, applications, and

tools to publish a suite of award-winning online services, including ESPN. com, ABCNEWS. com, Mr. Showbiz, NBA. com, and NFL. com, among others. Naughton is the author of The Java Handbook and coauthor of Java 1. : The Complete Reference, both best-sellers from Osborne/McGraw-Hill. He holds a B. S. in computersciencefrom Clarkson University. Herbert Schildt is the world's leading programming author. His books have sold more than two million copies worldwide and have been translated into all major foreign languages. Herb is author of the best-sellers C++: The Complete Reference, C: The Complete Reference, Java Programmer's Reference, Teach Yourself C, Teach Yourself C++, C++ from the Ground Up, Windows 98 Programming from the Ground Up, and STL Programming from the Ground Up. He is also a coauthor of C/C++ Annotated Archives.

Herb is president of Universal Computing Laboratories, a software consulting firm in Mahomet, Illinois. He holds a master's degree in computer science from the University of Illinois. Special Thanks -5- Special thanks go to Joe O'Neil for his help in preparing the third edition of this book. In addition to handling the updating required by the new Java 2 specification, Joe also provided the initial drafts for Chapters 24, 25, 26, and 27. As always, his efforts are appreciated. Acknowledgments Writing my first book, The Java Handbook, was a spiritual experience—bringing closure to my five years of effort on the Java project.

It reads more like a narrative tour through Java than a " complete reference. " Not all of the Java library classes were covered, and not every method in each class was listed. Culminating with the history chapter, " The Long

Strange Trip to Java," the book provided an outlet for my personal opinions about how the language turned out. This book is different. It presents a balanced, objective, and comprehensive view of Java. The heroics of the people who made this language happen are largely undocumented. The press tends to focus too narrowly in order to make for a clean story.

The success of this language is not due to any single person, but to the combined successes and failings of a group of dedicated and inspirational individuals—including James Gosling, Arthur van Hoff, Jonathan Payne, Chris Warth, Tim Lindholm, Frank Yellin, Sami Shaio, Patrick Chan, Kim Polese, Richard Tuck, Eugene Kuerner, Bill Joy, and many more. Myrespectfor what that team accomplished grows with each passing day and each Java class I write. The Internet is a strange place to work. For this book's first edition, I spent six months working closely with a man I've never met.

Herb Schildt wrote a wonderful book, C++: The Complete Reference, about a very difficult language. When it became clear that programmers needed a combination of what I offered in The Java Handbook and Herb's book, the solution was simple. We teamed up to provide the best of both worlds. Herb understands how to present difficult concepts in a way that neither insults the experienced programmer nor leaves the beginner behind. His stamina for writing down every little detail combined with my understanding of how those details came about has made for a book we think you will all enjoy.

We've updated this book for Java 2 so that it remains current and continues to deliver a solid reference in a proven format. Herb has done most of thehard workon this third edition. My main goal was to include complete

examples of excellent Java programming. To avoid presenting a single biased view about how to write Java, I also used some of my friends' excellent applets and they deserve credit for their work here: • Robert Temple is a gifted Java programmer who creates amazing applets with almost no download time.

I saw his code on the net and sent him e-mail to offer him a job, but coincidentally, he had found the job listing on the web and had already scheduled aninterviewtrip. His DynamicBillboard applet, which we examine in Chapter 29, is full of nonobvious performance tricks, which renewed my faith in Java the first time I saw it. • David LaVallee (a. k. a. Scout) is one of four people who were writing Java code in 1991. His ImageMenu and Lavatron applets in Chapters 30 and 31 are classic Scout design. Famous for his " baubles and trinkets," he always brings a creative design angle to his applets. David Geller is a longtime Windows programmer and author who has a keen eye for developer tools. His insight and contributions on development environments were invaluable. • Johanna Carr remains the smartest nonprogrammer in the world when it comes to software systems and languages. She diligently read and commented on every page of this book, at least twice. If Johanna can't understand it, it is probably poorly written. -6- Thanks also go to Matthew Naythons, who looks out for my best interests so I don't have to. And, to Kenna Moser, who continues to support my aspirations with love, encouragement, and great double lattes.

PATRICK J. NAUGHTON Preface Programming languages, paradigms, and practices don't stand still very long. It often seems that the methods and

techniques we applied yesterday are out-of-date today. Of course, this rapid rate of change is also one of the things that keeps programming exciting. There is always something new on the horizon. Perhaps no language better exemplifies the preceding statements than Java. In the space of just a few years, Java grew from a concept into one of the world's dominant computer languages. Moreover, during the same short period of time, Java has gone through two major revisions.

The first occurred when 1. 1 was released. The change in the minor revision number from 1. 0 to 1. 1 belies the significance of the 1. 1 specification. For example, Java 1. 1 fundamentally altered the way events were handled, added such features as Java Beans, and enhanced the API. The second major revision, Java 2, is the subject of this book. Java 2 keeps all of the functionality provided by Java 1. 1, but adds a substantial amount of new and innovative features. For example, it adds the collections framework, Swing, a new threading model, and numerous API methods and classes.

In fact, so many new features have been added that it is not possible to discuss them all in this book. In order to keep pace with Java, this book, too, has gone through rapid revision cycles. The original version of this book covered Java 1. 0. The second edition covered 1. 1. This, the third edition, covers Java 2. The time from the first edition to the third is less than two and one half years! But then, this book is about Java—and Java moves fast! A Book for All Programmers To use this book does not require any previous programming experience.

If, however, you come from a C/C++ background, then you will be able to advance a bit more rapidly. As most readers will know, Java is similar, in form and spirit, to C/C++. Thus, knowledge of those languages helps, but is not necessary. Even if you have never programmed before, you can learn to program in Java using this book. What's Inside This book covers all aspects of the Java programming language. Part I presents an indepth tutorial of the Java language. It begins with the basics, including such things as data types, control statements, and classes.

Part I also discusses Java's exceptionhandling mechanism, multithreading subsystem, packages, and interfaces. Part II examines the standard Java library. As you will learn, much of Java's power is found in its library. Topics include strings, I/O, networking, the standard utilities, the collections framework, applets, GUI-based controls, and imaging. Part III looks at some issues relating to the Java development environment, including an overview of Java Beans, Swing, and servlets. Part IV presents a number of high-powered Java applets, which serve as extended examples of the way Java can be applied.

The final applet, called Scrabblet, is a complete, -7- multiuser networked game. It shows how to handle some of the toughest issues involved in Web-based programming. What's New in the Third Edition The major differences between this and the previous editions of this book involve those features added by Java 2. These include such things as the collections framework, Swing, and the changes to the way multithreading is handled. However, there are also many smaller changes that are sprinkled throughout the Java

API. Another new item added to the book is the chapter on servlets, which are small programs that extend a Web server's functionality.

I think that you will find this to be a particularly interesting addition. A Team Effort I have been writing about programming for many years now. I seldom work with a coauthor. However, because of the special nature of this book, I teamed up with Patrick Naughton, one of the creators of Java. Patrick's insights, expertise, and energy contributed greatly to the success of this project. Because of Patrick's detailed knowledge of the Java language, its design, and implementation, there are tips and techniques found in this book that are difficult (if not impossible) to find elsewhere. HERBERT SCHILDT Part l: The Java Language

Chapter List Chapter 1: Chapter 2: Chapter 3: Chapter 4: Chapter 5: Chapter 6: Chapter 7: Chapter 8: Chapter 9: The Genesis of Java An Overview of Java Data Types, Variables, and Arrays Operators Control Statements Introducing Classes A Closer Look at Methods and Classes Inheritance Packages and Interfaces -8- Chapter 10: Chapter 11: Chapter 12: Exception Handling Multithreaded Programming I/O, Applets, and Other Topics Chapter 1: The Genesis of Java Overview When the chronicle of computer languages is written, the following will be said: B led to C, C evolved into C++, and C++ set the stage for Java.

To understand Java is to understand the reasons that drove its creation, the forces that shaped it, and the legacy that it inherits. Like the successful computer languages that came before, Java is a blend of the best elements of its rich heritage combined with the innovative concepts required by its

unique environment. While the remaining chapters of this book describe the practical aspects of Java—including its syntax, libraries, and applications—in this chapter, you will learn how and why Java came about, and what makes it so important.

Although Java has become inseparably linked with the online environment of the Internet, it is important to remember that Java is first and foremost a programming language. Computer language innovation and development occurs for two fundamental reasons: • To adapt to changing environments and uses • To implement refinements and improvements in the art of programming As you will see, the creation of Java was driven by both elements in nearly equal measure. Java's Lineage Java is related to C++, which is a direct descendent of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax.

Many of Java's object-oriented features were influenced by C++. In fact, several of Java's defining characteristics come from—or are responses to—its predecessors. Moreover, the creation of Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past three decades. For these reasons, this section reviews the sequence of events and forces that led up to Java. As you will see, each innovation in language design was driven by the need to solve a fundamental problem that the preceding languages could not solve. Java is no exception. The Birth of Modern Programming: C

The C language shook the computer world. Its impact should not be underestimated, because it fundamentally changed the way programming

was approached and thought about. The creation of C was a direct result of the need for a structured, efficient, highlevel language that could replace assembly code when creating systems programs. As you probably know, when a computer language is designed, trade-offs are often made, such as the following: • Ease-of-use versus power -9- • Safety versus efficiency • Rigidity versus extensibility Prior to C, programmers usually had to choose between languages that optimized one set of traits or the other.

For example, although FORTRAN could be used to write fairly efficient programs for scientific applications, it was not very good for systems code. And while BASIC was easy to learn, it wasn't very powerful, and its lack of structure made its usefulness questionable for large programs. Assembly language can be used to produce highly efficient programs, but it is not easy to learn or use effectively. Further, debugging assembly code can be quite difficult. Another compounding problem was that early computer languages such as BASIC, COBOL, and FORTRAN were not designed around structured principles.

Instead, they relied upon the GOTO as a primary means of program control. As a result, programs written using these languages tended to produce " spaghetti code"—a mass of tangled jumps and conditional branches that make a program virtually impossible to understand. While languages like Pascal are structured, they were not designed for efficiency, and failed to include certain features necessary to make them applicable to a wide range of programs. (Specifically, given the standard dialects of Pascal available at the time, it was not practical to consider using Pascal for systems-level code.

So, just prior to the invention of C, no one language had reconciled the conflicting attributes that had dogged earlier efforts. Yet the need for such a language was pressing. By the early 1970s, the computer revolution was beginning to take hold, and the demand for software was rapidly outpacing programmers' ability to produce it. A great deal of effort was being expended inacademiccircles in an attempt to create a better computer language. But, and perhaps most importantly, a secondary force was beginning to be felt. Computer hardware was finally becoming common enough that a critical mass was being reached.

No longer were computers kept behind locked doors. For the first time, programmers were gaining virtually unlimited access to their machines. This allowed the freedom to experiment. It also allowed programmers to begin to create their own tools. On the eve of C's creation, the stage was set for a quantum leap forward in computer languages. Invented and first implemented by Dennis Ritchie on a DEC PDP-11 running the UNIX operating system, C was the result of a development process that started with an older language called BCPL, developed by Martin Richards.

BCPL influenced a language called B, invented by Ken Thompson, which led to the development of C in the 1970s. For many years, the de facto standard for C was the one supplied with the UNIX operating system and described in The C Programming Language by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1978). C was formally standardized in December 1989, when the American National Standards Institute (ANSI) standard for C was adopted. The creation of C is considered by many to have marked the

beginning of the modern age of computer languages. It successfully synthesized the conflicting attributes that had so troubled earlier languages.

The result was a powerful, efficient, structured language that was relatively easy to learn. It also included one other, nearly intangible aspect: it was a programmer's language. Prior to the invention of C, computer languages were generally designed either as academic exercises or by bureaucratic committees. C is different. It was designed, implemented, and developed by real, working programmers, reflecting the way that they approached the job of programming. Its features were honed, tested, thought about, and rethought by the people who actually used the language.

The result was a language that programmers liked to use. Indeed, C quickly attracted many followers who had a near-religious zeal for it. As such, it found wide and rapid acceptance in the programmer community. In short, C is a language designed by and for programmers. As you will see, Java has inherited this legacy. The Need for C++ - 10 - During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today. Since C is a successful and useful language, you might ask why a need for something else existed.

The answer is complexity. Throughout the history of programming, the increasing complexity of programs has driven the need for better ways to manage that complexity. C++ is a response to that need. To better understand why managing program complexity is fundamental to the creation of C++, consider the following. Approaches to programming have changed dramatically since the invention of the computer. For example,

when computers were first invented, programming was done by manually toggling in the binary machine instructions by use of the front panel.

As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs by using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity. The first widespread language was, of course, FORTRAN. While FORTRAN was an impressive first step, it is hardly a language that encourages clear and easy-tounderstand programs.

The 1960s gave birth to structured programming. This is the method of programming championed by languages such as C. The use of structured languages enabled programmers to write, for the first time, moderately complex programs fairly easily. However, even with structured programming methods, once a project reaches a certain size, its complexity exceeds what a programmer can manage. By the early 1980s, many projects were pushing the structured approach past its limits. To solve this problem, a new way to program was invented, called object-oriented programming (OOP).

Object-oriented programming is discussed in detail later in this book, but here is a brief definition: OOP is a programming methodology that helps organize complex programs through the use of inheritance, encapsulation, and polymorphism. In the final analysis, although C is one of the world's great programming languages, there is a limit to its ability to handle complexity. Once a program exceeds somewhere between 25, 000 and 100,

000 lines of code, it becomes so complex that it is difficult to grasp as a totality. C++ allows this barrier to be broken, and helps the programmer comprehend and manage larger programs.

C++ was invented by Bjarne Stroustrup in 1979, while he was working at Bell Laboratories in Murray Hill, New Jersey. Stroustrup initially called the new language " C with Classes. " However, in 1983, the name was changed to C++. C++ extends C by adding object-oriented features. Because C++ is built upon the foundation of C, it includes all of C's features, attributes, and benefits. This is a crucial reason for the success of C++ as a language. The invention of C++ was not an attempt to create a completely new programming language. Instead, it was an enhancement to an already highly successful one.

C++ was standardized in November 1997, and an ANSI/ISO standard for C++ is now available. The Stage Is Set for Java By the end of the 1980s and the early 1990s, object-oriented programming using C++ took hold. Indeed, for a brief moment it seemed as if programmers had finally found the perfect language. Because C++ blended the high efficiency and stylistic elements of C with the object-oriented paradigm, it was a language that could be used to create a wide range of programs. However, just as in the past, forces were brewing that would, once again, drive computer language evolution forward. Within a few years, the World

Wide Web and the Internet would reach critical mass. This event would precipitate another revolution in programming. The Creation of Java - 11 - Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed

Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called " Oak" but was renamed " Java" in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language.

Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype. Somewhat surprisingly, the original impetus for Java was not the Internet! Instead, the primarymotivationwas the need for a platform-independent (that is, architecture- neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. As you can probably guess, many different types of CPUs are used as controllers.

The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier—and more cost-efficient —solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments.

This effort ultimately led to the creation of Java. About the time that the details of Java were being worked out, a second, and ultimately more important, factor was emerging that would play a crucial role in the future of Java. This second force was, of course, the World Wide Web. Had the Web not taken shape at about the same time that Java was being implemented, Java might have remained a useful but obscure language for programming consumer electronics.

However, with the emergence of the World Wide Web, Java was propelled to the forefront of computer language design, because the Web, too, demanded portable programs. Most programmers learn early in their careers that portable programs are as elusive as they are desirable. While the quest for a way to create efficient, portable (platformindependent) programs is nearly as old as the discipline of programming itself, it had taken a back seat to other, more pressing problems.

Further, because much of the computer world had divided itself into the three competing camps of Intel, Macintosh, and UNIX, most programmers stayed within their fortified boundaries, and the urgent need for portable code was reduced. However, with the advent of the Internet and the Web, the old problem of portability returned with a vengeance. After all, the Internet consists of a diverse, distributed universe populated with many types of computers, operating systems, and CPUs. Even though many types of platforms are attached to the Internet, users would like them all to be able to run the same program.

What was once an irritating but low-priority problem had become a high-profile necessity. By 1993, it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet. In fact, the same problem that Java was initially designed to solve on a small scale could also be applied to the Internet on a large scale. This realization caused the focus of Java to switch from consumer electronics to Internet programming.

So, while the desire for an architecture-neutral programming language provided the initial spark, the Internet ultimately led to Java's large-scale success. As mentioned earlier, Java derives much of its character from C and C++. This is by intent. The Java designers knew that using the familiar syntax of C and echoing the object-oriented features of C++ would make their language appealing to the legions of experienced C/C++ programmers. In addition to the surface similarities, Java shares some of the other attributes that helped make C and C++ successful.

First, Java was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and experiences of the people who devised it. Thus, Java is also a programmer's language. Second, Java is cohesive and logically consistent. Third, except for those constraints imposed by the Internet environment, Java gives you, the - 12 - programmer, full control. If you program well, your programs reflect it. If you program poorly, your programs reflect that, too. Put differently, Java is not a language with training wheels. It is a language for professional programmers.

Because of the similarities between Java and C++, it is tempting to think of Java as simply the " Internet version of C++. " However, to do so would be a large mistake. Java has significant practical and philosophical differences. While it is true that Java was influenced by C++, it is not an enhanced version of C++. For example, Java is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a C++ programmer, then you will feel right at home with Java. One other point: Java was not designed to replace C++.

Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems. Both will coexist for many years to come. As mentioned at the start of this chapter, computer languages evolve for two reasons: to adapt to changes in environment and to implement advances in the art of programming. The environmental change that prompted Java was the need for platform-independent programs destined for distribution on the Internet. However, Java also embodies changes in the way that people approach the writing of programs.

Specifically, Java enhances and refines the object-oriented paradigm used by C++. Thus, Java is not a language that exists in isolation. Rather, it is the current instance of an ongoing process begun many years ago. This fact alone is enough to ensure Java a place in computer language history. Java is to Internet programming what C was to systems programming: a revolutionary force that will change the world. Why Java Is Important to the Internet The Internet helped catapult Java to the forefront of programming, and Java, in turn, has had a profound effect on the Internet.

The reason for this is quite simple: Java expands the universe of objects that can move about freely in cyberspace. In a network, two very broad categories of objects are transmitted between the server and your personal computer: passive information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. However, a second type of object can be transmitted to your computer: a dynamic, self-executing program.

Such a program is an active agent on the client computer, yet is initiated by the server. For example, a program might be provided by the server to display properly the data that the server is sending. As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Prior to Java, cyberspace was effectively closed to half the entities that now live there. As you will see, Java addresses those concerns and, by doing so, has opened the door to an exciting new form of program: the applet. Java Applets and Applications

Java can be used to create two types of programs: applications and applets. An application is a program that runs on your computer, under the operating system of that computer. That is, an application created by Java is more or less like one created using C or C++. When used to create applications, Java is not much different from any other computer language. Rather, it is Java's ability to create applets that makes it important. An applet is an application designed to be transmitted over the Internet and executed by a Java-compatible Web browser.

An applet is actually a tiny Java program, dynamically downloaded across the network, just like an image, sound file, or video clip. The important difference is that an applet is an intelligent program, not just an animation or media file. In other words, an applet is a program that can react to user input and dynamically change—not just run the same animation or sound over and over. As exciting as applets are, they would be nothing more than wishful thinking if Java were not able to address the two fundamental problems associated with them: security and portability.

Before continuing, let's define what these two terms mean relative to the - 13 - Internet. Security As you are likely aware, every time that you download a " normal" program, you are risking a viral infection. Prior to Java, most users did not download executable programs frequently, and those who did scanned them for viruses prior to execution. Even so, most users still worried about the possibility of infecting their systems with a virus. In addition to viruses, another type of malicious program exists that must be guarded against.

This type of program can gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. Java answers both of these concerns by providing a " firewall" between a networked application and your computer. When you use a Java-compatible Web browser, you can safely download Java applets without fear of viral infection or malicious intent. Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer. You will see how

this is accomplished shortly.) The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most important aspect of Java. Portability As discussed earlier, many types of computers and operating systems are in use throughout the world—and many are connected to the Internet. For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed.

As you will soon see, the same mechanism that helps ensure security also helps create portability. Indeed, Java's solution to these two problems is both elegant and efficient. Java's Magic: The Bytecode The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM). That is, in its standard form, the

JVM is an interpreter for bytecode. This may come as a bit of a surprise. As you know, C++ is compiled to executable code. In fact, most modern languages are designed to be compiled, not interpreted—mostly because of performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with downloading programs over the Internet. Here is why. Translating a Java program into bytecode helps makes it much easier to run a program in a wide variety of

environments. The reason is straightforward: only the JVM needs to be implemented for each platform.

Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all interpret the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the interpretation of bytecode is the easiest way to create truly portable programs. The fact that a Java program is interpreted also helps to make it secure.

Because the execution of every Java program is under the control of the JVM, the JVM can contain the program and prevent it from generating side effects outside of the system. As you will see, safety is also enhanced by certain restrictions that exist in the Java language. When a program is interpreted, it generally runs substantially slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. The use of bytecode enables the Java run-time system to execute programs much faster than you might expect. - 14 -

Although Java was designed for interpretation, there is technically nothing about Java that prevents on-the-fly compilation of bytecode into native code. Along these lines, Sun has just completed its Just In Time (JIT) compiler for bytecode, which is included in the Java 2 release. When the JIT compiler is part of the JVM, it compiles bytecode into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not

possible to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be one only at run time. Instead, the JIT compiles code as it is needed, during execution. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the run-time system (which performs the compilation) still is in charge of the execution environment. Whether your Java program is actually interpreted in the traditional way or compiled on-the-fly, its functionality is the same. The Java Buzzwords No discussion of the genesis of Java is complete without a look at the Java buzzwords.

Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in molding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords: • Simple • Secure • Portable • Object-oriented • Robust • Multithreaded • Architecture-neutral • Interpreted • High performance • Distributed • Dynamic Two of these buzzwords have already been discussed: secure and portable. Let's examine what each of the others implies. Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many

of the object-oriented features of C++, most programmers have little trouble learning Java. Also, some of the more confusing concepts from C++ are 15 - either left out of Java or implemented in a cleaner, more approachable manner. Beyond its similarities with C/C++, Java has another attribute that makes it easy to learn: it makes an effort not to have surprising features. In Java, there are a small number of clearly defined ways to accomplish a given task. Object-Oriented Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects.

Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's " everything is an object" paradigm and the pragmatist's " stay out of my way" model. The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as highperformance nonobjects. Robust The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java.

To gain reliability, Java restricts you in a few key areas, to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. In

fact, many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

To better understand how Java is robust, consider two of the main reasons for programfailure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using.

Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or " file not found," and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should — be managed by your program. Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an

elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem. Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a - 16 - program today, it will run tomorrow —even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was " write once; run anywhere, any time, forever. To a great extent, this goal was accomplished. Interpreted and High Performance As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be interpreted on any system that provides a Java Virtual Machine. Most previous attempts at crossplatform solutions have done so at the expense of performance. Other interpreted systems, such as BASIC, Tcl, and PERL, suffer from almost insurmountable performance deficits. Java, however, was designed to perform well on very low-power CPUs.

As explained earlier, while it is true that Java was engineered for interpretation, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high

performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code. " High-performance cross-platform" is no longer an oxymoron. Distributed Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols.

In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intra-addressspace messaging. This allowed objects on two different computers to execute procedures remotely. Java has recently revived these interfaces in a package called Remote Method Invocation (RMI). This feature brings an unparalleled level of abstraction to client/server programming. Dynamic Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.

This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the applet environment, in which small fragments of bytecode may be dynamically updated on a running system. The Continuing Revolution The initial release of Java was nothing short of revolutionary, but it did not mark the end of Java's era of rapid innovation. Unlike most other software systems that usually settle into a pattern of small, incremental improvements, Java continued to evolve at an explosive pace. Soon after the release of Java 1. , the designers of Java had already created Java 1. 1. The features added by Java 1. 1 were more significant and substantial than the increase in the minor revision number would have you think. Java 1. 1 added many new library elements, redefined

the way events are handled by applets, and reconfigured many features of the 1. 0 library. It also deprecated (rendered obsolete) several features originally defined by Java 1. 0. Thus, Java 1. 1 both added and subtracted attributes from its original specification. Continuing in this evolution, Java 2 also adds and subtracts features.

While all languages change over time, changes to Java take on an extra importance, because older browsers will not be able to execute code that uses a new feature. For this reason, it is good to have a general understanding of when various changes have taken place. With this in mind, the next section takes a brief look at the evolution of Java since its original 1. 0 specification. - 17 - Features Added by 1. 1 Version 1. 1 added some important elements to Java. Most of the additions occurred in the Java library. However, a few new language features were also included. Here is a list of the most important features added by 1. : • Java Beans, which are software components that are written in Java. • Serialization, which allows you to save and restore the state of an object. • Remote Method Invocation (RMI), which allows a Java object to invoke the methods of another Java object that is located on a different machine. This is an important facility for building distributed applications. • Java Database Connectivity (JDBC), which allows programs to access SQL databases from many different vendors. • The Java Native Interface (JNI), which provides a new way for your programs to interface with code libraries written in other languages. Reflection, which is the process of determining the fields, constructors, and methods of a Java object at run time. • Various security features, such as digital signatures, message digests, access control lists, and key generation. • Built-in support

for 16-bit character streams that handle Unicode characters. • Significant changes to event handling that improve the way in which events generated by graphical user interface (GUI) components are handled. • Inner classes, which allow one class to be defined within another. Features Deprecated by 1. 1 As just mentioned, Java 1. 1 deprecated many earlier library elements.

For example, most of the original Date class was deprecated. However, the deprecated features did not go away. Instead, they were replaced with updated alternatives. In general, deprecated 1. 0 features are still available in Java to support legacy code, but they should not be used by new applications. This book still describes a few of the more important deprecated 1. 0 library elements, for the sake of programmers older code. Features Added by 2 Building upon 1. 1, Java 2 adds many important new features. Here is a partial list: • Swing is a set of user interface components that is implemented entirely in Java.

You can use a look and feel that is either specific to a particular operating system or uniform across operating systems. You can also design your own look and feel. • Collections are groups of objects. Java 2 provides several types of collections, such as linked lists, dynamic arrays, and hash tablefsections, for your use. Collections offer a new way to solve several common programming problems. • More flexible security mechanisms are now available for Java programs. Policy files can define the permissions for code from various sources. These determine, for example, whether a particular file or directory may be accessed, or whether a 18 - connection can be established to a specific host and port. • Digital certificates provide a

mechanism to establish the identity of a user. You may think of them as electronic passports. Java programs can parse and use certificates to enforce security policies. • Various security tools are available that enable you to create and store cryptographic keys and digital certificates, sign Java Archive (JAR) files, and check the signature of a JAR file. • The Accessibility library provides features that make it easier for people with sight impairments or other disabilities to work with computers.

Of course, these capabilities can be useful for any user. • The Java 2D library provides advanced features for working with shapes, images, and text. • Drag-and-drop capabilities allow you to transfer data within or between applications. • Text components can now receive Japanese, Chinese, and Korean characters from the keyboard. This is done by using a sequence of keystrokes to represent one character. • You can now play back WAV, AIFF, AU, MIDI, and RMF audio files. • The Common Object Request Broker Architecture (CORBA) defines an Object Request Broker (ORB) and an Interface Definition Language (IDL).

Java 2 includes an ORB and an idltojava compiler. The latter generates code from an IDL specification. • Performance improvements have been made in several areas. A Just-In-Time (JIT) compiler is included in the JDK. • Many browsers include a Java Virtual Machine that is used to execute applets. Unfortunately, browser JVMs typically do not include the latest Java features. The Java Plug-In solves this problem. It directs a browser to use a Java Runtime Environment (JRE) rather than the browser's JVM. The JRE is a subset of the JDK. It does not include the tools and classes that are used in a

development environment. Various tools such as javac, java, and javadoc have been enhanced. Debugger and profiler interfaces for the JVM are available. Features Deprecated by 2 Although not as extensive as the deprecations experienced between 1. 0 and 1. 1, some features of Java 1. 1 are deprecated by Java 2. For example, the suspend( ), resume( ), and stop( ) methods of the Thread class should not be used in new code. Throughout this book, deprecated features are pointed out and their Java 2 alternatives are described. This will be helpful to any programmer charged with updating Java 1. 1 code. Java Is Not an Enhanced HTML

Before moving on, it is necessary to dispel a common misunderstanding. Because Java is used in the creation of Web pages, newcomers sometimes confuse Java with Hypertext Markup Language (HTML) or think that Java is simply some enhancement to HTML. Fortunately, these are misconceptions. HTML is, in essence, a means of defining the logical organization of information and providing links, called hypertext links, to related information. As you probably know, a hypertext link (also called a hyperlink) is a link to another hypertext document, which may exist either locally or elsewhere on the Web.

The defining element of a hypertext document is that it can be read in a nonlinear - 19 - fashion, with the user pursuing various paths by choosing hypertext links to other, related documents. Although HTML allows a user to read documents in a dynamic manner, HTML is not, and never has been, a programming language. While it is certainly true that, to the extent that HTML helped propel the popularity of the Web, HTML was a catalyst for the

creation of Java, it did not directly influence the design of the language or the concepts behind it.

The only connection that HTML has to Java is that it provides the applet tag, which executes a Java applet. Thus, it is possible to embed instructions in a hypertext document that cause a Java applet to execute. Chapter 2: An Overview of Java Overview Like all other computer languages, the elements of Java do not exist in isolation. Rather, they work together to form the language as a whole. However, this interrelatedness can make it difficult to describe one aspect of Java without involving several others. Often a discussion of one feature implies prior knowledge of another.

For this reason, this chapter presents a quick overview of several key features of Java. The material described here will give you a foothold that will allow you to write and understand simple programs. Most of the topics discussed will be examined in greater detail in the remaining chapters of Part 1. Object-Oriented Programming Object-oriented programming is at the core of Java. In fact, all Java programs are objectoriented—this isn't an option the way that it is in C++, for example. OOP is so integral to Java that you must understand its basic principles before you can write even simple Java programs.

Therefore, this chapter begins with a discussion of the theoretical aspects of OOP. Two Paradigms As you know, all computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around " what is happening" and others are written around " who is

being affected. " These are the two paradigms that govern how a program is constructed. The first way is called the process-oriented model. This approach characterizes a program as a series of linear steps (that is, code).

The process-oriented model can be thought of as code acting on data. Procedural languages such as C employ this model to considerable success. However, as mentioned in Chapter 1, problems with this approach appear as programs grow larger and more complex. To manage increasing complexity, the second approach, called object-oriented programming, was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An objectoriented program can be characterized as data controlling access to code.

As you will see, by switching the controlling entity to data, you can achieve several organizational benefits. Abstraction An essential element of object-oriented programming is abstraction. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car.

They can ignore the details of how the engine, transmission, and braking systems work. Instead - 20 - they are free to utilize the object as a whole. A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a

single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units.

For instance, the sound system consists of a radio, a CD player, and/or a tape player. The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions. Hierarchical abstractions of complex systems can also be applied to computer programs. The data from a traditional process-oriented program can be transformed by abstraction into its component objects. A sequence of process steps can become a collection of messages between these objects. Thus, each of these objects describes its own unique behavior.

You can treat these objects as concrete entities that respond to messages telling them to do something. This is the essence of object-oriented programming. Object-oriented concepts form the heart of Java just as they form the basis for human understanding. It is important that you understand how these concepts translate into programs. As you will see, object-oriented programming is a powerful and natural paradigm for creating programs that survive the inevitable changes accompanying the life cycle of any major software project, including conception, growth, and aging.

For example, once you have well-defined objects and clean, reliable interfaces to those objects, you can gracefully decommission or replace parts of an older system without fear. The Three OOP Principles All object-oriented programming languages provide mechanisms that help you implement the

object-oriented model. They are encapsulation, inheritance, and polymorphism. Let's take a look at these concepts now. Encapsulation Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever.

You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever. You can't affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please.

However, from the driver's point of view, they all work the same. This same idea can be applied to programming. The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects. In Java

the basis of encapsulation is the class. Although the class will be examined in great detail later in this book, the following brief discussion will be helpful now. A class defines the structure and behavior (data and code) that will be shared by a set of objects.

Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as instances of a class. Thus, a class is a logical construct; an object has physical reality. - 21 - When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called members of the class. Specifically, the data defined by the class are referred to as member variables or instance variables.

The code that operates on that data is referred to as member methods or just methods. (If you are familiar with C/C++, it may help to know that what a Java programmer calls a method, a C/C++ programmer calls a function.) In properly written Java programs, the methods define how the member variables can be used. This means that the behavior and interface of a class are defined by the methods that operate on its instance data. Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class.

Each method or variable in a class may be marked private or public. The public interface of a class represents everything that external users of the class need to know, or may know. The private methods and data can only be accessed by code that is a member of the class. Therefore, any other code

that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class' pu