THE EXPERT'S VOICE® IN JAVA

# Beginning Java 7

Jeff Friesen

Apress®

# Beginning Java 7

■ ■ ■

Jeff Friesen

Apress®

**Beginning Java 7**

Copyright © 2011 by Jeff Friesen

ISBN-13 (pbk): 978-1-4302-3909-3

ISBN-13 (electronic): 978-1-4302-3910-9

Distributed to the book trade worldwide by Springer Science+Business Media New York, LLC., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/bulk-sales`.

The source code for this book is available to readers at `www.apress.com`. You will need to answer questions pertaining to this book in order to successfully download the code.

# Contents at a Glance

# Contents

# About the Author

■ **Jeff Friesen** is a freelance tutor and software developer with an emphasis on Java (and now Android). Besides writing this book, Jeff has authored Apress's *Learn Java for Android Development* (ISBN13: 978-1-4302-3156-1), has coauthored Apress's *Android Recipes* (ISBN13: 978-1-4302-3413-5) with Dave Smith, and has written numerous articles on Java and other technologies for Java.net (`www.java.net`), JavaWorld (`www.javaworld.com`), InformIT (`www.informit.com`), and DevSource (`www.devsource.com`). Jeff can be contacted via his TutorTutor website at `tutortutor.ca`.

# About the Technical Reviewer

■ Chád Darby is an author, instructor, and speaker in the Java development world. As a recognized authority on Java applications and architectures, he has presented technical sessions at software development conferences worldwide. In his 15 years as a professional software architect, he's had the opportunity to work for Blue Cross/Blue Shield, Merck, Boeing, Northrop Grumman, and various IT companies.

Chád is a contributing author to several Java  books, including *Professional Java E-Commerce* (Wrox Press), *Beginning Java Networking* (Wrox Press), and *XML and Web Services Unleashed* (Sams Publishing). He is also the author of numerous magazine articles for the Java Developer's Journal (Sys-Con Publishing).

Chád has Java certifications from Sun Microsystems and IBM. He holds a B.S. in Computer Science from Carnegie Mellon University. In his free time, Chád enjoys running half-marathons..

# Acknowledgments

*Beginning Java 7* wouldn't have been possible without the wonderful folks at Apress. I thank Steve Anglin for giving me the opportunity to write this book, Corbin Collins for guiding me through the various aspects of the book-writing process, Tom Welsh for helping me with the development of my chapters and appendixes, and Chad Darby for his diligence in catching various flaws. I couldn't ask for better editors and a better technical reviewer. Thanks guys.

# Introduction

Java 7 is Oracle's latest release of the popular Java language and platform. *Beginning Java 7* guides you through this language and a huge assortment of platform APIs via its 12 chapters and 4 appendixes.

**Note** Java was created by Sun Microsystems, which was later bought out by Oracle.

Chapter 1 (Getting Started with Java) introduces you to Java and begins to cover the Java language by focusing on fundamental concepts such as comments, identifiers, variables, expressions, and statements.

Chapter 2 (Discovering Classes and Objects) continues to explore this language by presenting all of its features for working with classes and objects. You learn about features related to class declaration and object creation, encapsulation, information hiding, inheritance, polymorphism, interfaces, and garbage collection.

Chapter 3 (Exploring Advanced Language Features) focuses on the more advanced language features related to nested classes, packages, static imports, exceptions, assertions, annotations, generics, and enums. Subsequent chapters introduce you to the few features not covered in Chapters 1 through 3.

Chapter 4 (Touring Language APIs) largely moves away from covering language features (although it does introduce class literals and strictfp) while focusing on language-oriented APIs. You learn about Math, StrictMath, Package, Primitive Type Wrapper Classes, Reference, Reflection, String, StringBuffer and StringBuilder, Threading, BigDecimal, and BigInteger in this chapter.

Chapter 5 (Collecting Objects) begins to explore Java's utility APIs by focusing largely on the Collections Framework. However, it also discusses legacy collection-oriented APIs and how to create your own collections.

Chapter 6 (Touring Additional Utility APIs) continues to focus on utility APIs by presenting the concurrency utilities along with the Objects and Random classes.

Chapter 7 (Creating and Enriching Graphical User Interfaces) moves you away from the command-line user interfaces that appear in previous chapters and toward graphical user interfaces. You first learn about the Abstract Window Toolkit foundation and then explore the Java Foundation Classes in terms of Swing and Java 2D. (Appendix C introduces you to Accessibility and Drag and Drop.)

Chapter 8 (Interacting with Filesystems) explores filesystem-oriented I/O in terms of the File, RandomAccessFile, stream, and writer/reader classes. (New I/O is covered in Appendix C.)

Chapter 9 (Interacting with Networks and Databases) introduces you to Java's network APIs (e.g., sockets). It also introduces you to the JDBC API for interacting with databases.

Chapter 10 (Parsing, Creating, and Transforming XML Documents) dives into Java's XML support by first presenting an introduction to XML (including DTDs and schemas). It next explores the SAX, DOM, StAX, XPath, and XSLT APIs; and even briefly touches on the Validation API. While exploring XPath, you encounter namespace contexts, extension functions and function resolvers, and variables and variable resolvers.

Chapter 11 (Working with Web Services) introduces you to Java's support for SOAP-based and RESTful web services. Besides providing you with the basics of these web service categories, Chapter 11 presents some advanced topics, such as working with the SAAJ API to communicate with a SOAP-based web service without having to rely on JAX-WS. You'll appreciate having learned about XML in Chapter 10 before diving into this chapter.

Chapter 12 (Java 7 Meets Android) helps you put to use some of the knowledge you've gathered in previous chapters by showing you how to use Java to write an Android app's source code. This chapter introduces you to Android, discusses its architecture, shows you how to install necessary tools, and develops a simple app.

As well as creating these twelve chapters, I've created four appendices:

Appendix A (Solutions to Exercises) presents the solutions to the programming exercises that appear near the end of Chapters 1 through 12.

Appendix B (Scripting API and Dynamically Typed Language Support) introduces you to Java's Scripting API along with the support for dynamically typed languages that's new in Java 7.

Appendix C (Odds and Ends) introduces you to additional APIs and architecture topics: Accessibility, ByteArrayOutputStream and ByteArrayInputStream, classloaders, Console, Desktop, Drag and Drop, Dynamic Layout, Extension Mechanism and ServiceLoader, File Partition-Space, File Permissions, Formatter, Image I/O, Internationalization, Java Native Interface, NetworkInterface and InterfaceAddress, New I/O (including NIO.2), PipedOutputStream and PipedInputStream, Preferences, Scanner, Security, Smart Card, Splash Screen, StreamTokenizer, StringTokenizer, SwingWorker, System Tray, Timer and TimerTask, Tools and the Compiler API, Translucent and Shaped Windows, and XML Digital Signature.

Appendix D (Applications Gallery) presents a gallery of significant applications that demonstrate various aspects of Java and gives you an opportunity to have more fun with this technology.

Unfortunately, there are limits to how much knowledge can be crammed into a print book. For this reason, Appendixes A, B, C, and D are not included in this book's pages–adding these appendixes would have exceeded the Print-On-Demand (`http://en.wikipedia.org/wiki/Print_on_demand`) limit of 1,000 pages cover to cover. Instead, these appendixes are freely distributed as PDF files. Appendixes A and B are bundled with the book's associated code file at the Apress website (`http://www.apress.com/9781430239093`). Appendixes C and D are bundled with their respective code files on my TutorTutor website (`http://tutortutor.ca/cgi-bin/makepage.cgi?/books/bj7`).

Appendixes C and D are "living documents" in that I'll occasionally add new material to them. When I first encountered Java, I fell in love with this technology and dreamed about writing a book that explored the entire language and all standard edition APIs. Perhaps I would be the first person to do so.

There are various obstacles to achieving this goal. For one thing, it's not easy to organize a vast amount of content, and Java keeps getting bigger with each new release, so there's always more to write about.

Another obstacle is that it's not possible to adequately cover everything within the limits of a 1,000-page book. And then there are the time constraints, which make it impossible to complete everything in just a few months.

Proper organization is essential to creating a book that satisfies both Java beginners and more seasoned Java developers. Regrettably, lack of proper organization in my former *Learn Java for Android Development* book resulted in something that isn't beginner friendly (this has been pointed out on numerous occasions). For example, the second chapter mixes coverage of basic features (e.g., expressions and statements) with objects and classes, and this approach is too confusing for the novice. *Beginning Java 7*'s coverage of the Java language is better organized.

It's not possible to cover everything within 1,000 pages, which is the upper limit for a Print-On-Demand book. For this reason, I've designed Appendixes C and D to be "living" extensions to the book. They make it possible for me to complete my coverage of the entire Java 7 Standard Edition. I might even cover Java 8's new features in a separate area of Appendix C.

I spent nearly six months writing *Beginning Java 7*. Given the vast scope of this project, that's a very small amount of time. It will take me many more months to complete my tour of Java 7 Standard Edition; I'll occasionally post updated Appendixes C and D on my website that take you even deeper into this technology.

If you've previously purchased a copy of *Learn Java for Android Development*, you'll probably be shocked to discover that I've plagiarized much of my own content. I did so to speed *Beginning Java 7*'s development, which contains much material beyond what appeared in my former book (e.g., Swing and web services). *Beginning Java 7* would have taken many more months to complete if I didn't leverage its predecessor. (If I thought that *Learn Java for Android Development* was crap, and I don't, I never would have used it as the basis for this new book.)

Don't get the idea that *Beginning Java 7* is a rehash of *Learn Java for Android Development*–it's not. In those portions of *Beginning Java 7* where I've stolen heavily from its predecessor, there typically are numerous changes and additions. For example, I've rewritten parts of the exceptions and generics content that appear in Chapter 3; I did so to introduce new Java 7 features and to provide better coverage of difficult topics. Also, Chapter 5 introduces navigable sets and navigable maps, which is something that I couldn't discuss in *Learn Java for Android Development* because these features were introduced in Java 6. (I wrote *Learn Java for Android Development* to teach the Java language and APIs to prepare the reader for Android–Android apps are written in Java. However, Android doesn't support language features and APIs beyond Java 5.)

*Beginning Java 7* goes far beyond *Learn Java for Android Development* in that it also discusses user interface APIs (e.g., Abstract Window Toolkit, Swing, and Java 2D) and web services (JAX-WS and RESTful). As well as new content, you'll also find many new examples (e.g., a chat server) and new exercises (e.g., create a networked Blackjack game with a graphical user interface).

At the end of Chapter 10 in *Learn Java for Android Development*, I rashly promised to write the following free chapters:

Chapter 11: Performing I/O Redux

Chapter 12: Parsing and Creating XML Documents

Chapter 13: Accessing Networks

Chapter 14: Accessing Databases

Chapter 15: Working with Security

Chapter 16: Odds and Ends

I originally intended to write these chapters and add them to *Learn Java for Android Development*. However, I ran out of time and would probably have also run into the Print-On-Demand limit that I previously mentioned.

Given beginner-oriented organizational difficulties with *Learn Java for Android Development*, I decided to not write these chapters in that book's context. Instead, I pursued *Beginning Java 7* in a new (and hopefully better organized) attempt to cover all of Java, and to attempt to create a book that broadly appeals to Java beginners and veterans alike.

Although I won't write the aforementioned six free chapters as described in *Learn Java for Android Development* (I can't keep the entire promise anyway because I've integrated Chapters 12, 13, and 14 into *Beginning Java 7* as Chapters 9 and 10), the other three chapters (11, 15, and 16) are merged into Appendix C, which is free. As time passes, additional chapters will appear in that appendix; and so I will finally keep my promise, but in a different way.

---

■ **Note** I don't discuss code conventions for writing source code in this book. Instead, I've adopted my own conventions, and try to apply them consistently throughout the book. If you're interested in what Oracle has to say about Java code conventions, check out the "Code Conventions for the Java Programming Language" document at `http://www.oracle.com/technetwork/java/codeconv-138413.html`.

---

CHAPTER 1

# Getting Started with Java

Welcome to Java. This chapter launches you on a tour of this technology by focusing on fundamentals. First, you receive an answer to the "What is Java?" question. If you have not previously encountered Java, the answer might surprise you. Next, you are introduced to some basic tools that will help you start developing Java programs, and to the NetBeans integrated development environment, which simplifies the development of these programs. Finally, you explore fundamental language features.

## What Is Java?

Java is a language for describing programs, and Java is a platform on which to run programs written in Java and other languages (e.g., Groovy, Jython, and JRuby). This section introduces you to Java the language and Java the platform.

■ **Note** To discover Java's history, check out Wikipedia's "Java (programming language)" (`http://en.wikipedia.org/wiki/Java_(programming_language)#History`) and "Java (software platform)" (`http://en.wikipedia.org/wiki/Java_(software_platform)#History`) entries.

### Java Is a Language

*Java* is a general-purpose, class-based, and object-oriented language patterned after C and C++ to make it easier for existing C/C++ developers to migrate to this language. Not surprisingly, Java borrows elements from these languages. The following list identifies some of these elements:

- Java supports the same single-line and multiline comment styles as found in C/C++ for documenting source code.
- Java provides the `if`, `switch`, `while`, `for`, and other reserved words as found in the C and C++ languages. Java also provides the `try`, `catch`, `class`, `private`, and other reserved words that are found in C++ but not in C.
- As with C and C++, Java supports character, integer, and other primitive types. Furthermore, Java shares the same reserved words for naming these types; for example, `char` (for character) and `int` (for integer).

- Java supports many of the same operators as C/C++: the arithmetic operators (`+`, `-`, `*`, `/`, and `%`) and conditional operator (`?:`) are examples.
- Java also supports the use of brace characters { and } to delimit blocks of statements.

Although Java is similar to C and C++, it also differs in many respects. The following list itemizes some of these differences:

- Java supports an additional comment style known as Javadoc.
- Java provides `transient`, `synchronized`, `strictfp`, and other reserved words not found in C or C++.
- Java's character type has a larger size than the version of this type found in C and C++, Java's integer types do not include unsigned variants of these types (Java has no equivalent of the C/C++ unsigned long integer type, for example), and Java's primitive types have guaranteed sizes, whereas no guarantees are made for the equivalent C/C++ types.
- Java doesn't support all of the C/C++ operators. For example, there is no `sizeof` operator. Also, Java provides some operators not found in C/C++. For example, `>>>` (unsigned right shift) and `instanceof` are exclusive to Java.
- Java provides labeled break and continue statements. These variants of the C/C++ break and continue statements provide a safer alternative to C/C++'s goto statement, which Java doesn't support.

---

■ **Note** Comments, reserved words, types, operators, and statements are examples of fundamental language features, which are discussed later in this chapter.

---

A Java program starts out as source code that conforms to Java *syntax*, rules for combining symbols into meaningful entities. The Java compiler translates the source code stored in files that have the "`.java`" file extension into equivalent executable code, known as *bytecode*, which it stores in files that have the "`.class`" file extension.

---

■ **Note** The files that store compiled Java code are known as *classfiles* because they often store the runtime representation of Java classes, a language feature discussed in Chapter 2.

---

The Java language was designed with portability in mind. Ideally, Java developers write a Java program's source code once, compile this source code into bytecode once, and run the bytecode on any platform (e.g., Windows, Linux, and Mac OS X) where Java is supported, without ever having to change the source code and recompile. Portability is achieved in part by ensuring that primitive types have the same sizes across platforms. For example, the size of Java's integer type is always 32 bits.

The Java language was also designed with robustness in mind. Java programs should be less vulnerable to crashes than their C/C++ counterparts. Java achieves robustness in part by not implementing certain C/C++ features that can make programs less robust. For example, *pointers* (variables that store the addresses of other variables) increase the likelihood of program crashes, which is why Java doesn't support this C/C++ feature.

## Java Is a Platform

*Java* is a platform that executes Java-based programs. Unlike platforms with physical processors (e.g., an Intel processor) and operating systems (e.g., Windows 7), the Java platform consists of a virtual machine and execution environment.

A *virtual machine* is a software-based processor with its own set of instructions. The Java Virtual Machine (JVM)'s associated *execution environment* consists of a huge library of prebuilt functionality, commonly known as the *standard class library*, that Java programs can use to perform routine tasks (e.g., open a file and read its contents). The execution environment also consists of "glue" code that connects the JVM to the underlying operating system.

---

■ **Note** The "glue" code consists of platform-specific libraries for accessing the operating system's windowing, networking, and other subsystems. It also consists of code that uses the Java Native Interface (JNI) to bridge between Java and the operating system. I discuss the JNI in Appendix C. You might also want to check out Wikipedia's "Java Native Interface" entry (`http://en.wikipedia.org/wiki/Java_Native_Interface`) to learn about the JNI.

---

When a Java program launcher starts the Java platform, the JVM is launched and told to load a Java program's starting classfile into memory, via a component known as a *classloader*. After the classfile has loaded, the following tasks are performed:

- The classfile's bytecode instruction sequences are verified to ensure that they don't compromise the security of the JVM and underlying environment. Verification ensures that a sequence of instructions doesn't find a way to exploit the JVM to corrupt the environment and possibly steal sensitive information. The component that handles this task is known as the *bytecode verifier*.
- The classfile's main sequence of bytecode instructions is executed. The component that handles this task is known as the *interpreter* because instructions are *interpreted* (identified and used to select appropriate sequences of native processor instructions to carry out the equivalent of what the bytecode instructions mean). When the interpreter discovers that a bytecode instruction sequence is executed repeatedly, it informs the *Just-In-Time (JIT) compiler* component to compile this sequence into an equivalent sequence of native instructions. The JIT helps the Java program achieve faster execution than would be possible through interpretation alone. Note that the JIT and the Java compiler that compiles source code into bytecode are two separate compilers with two different goals.

During execution, a classfile might refer to another classfile. In this situation, a classloader is used to load the referenced classfile, the bytecode verifier then verifies the classfile's bytecodes, and the interpreter/JIT executes the appropriate bytecode sequence in this other classfile.

The Java platform was designed with portability in mind. By providing an abstraction over the underlying operating system, bytecode instruction sequences should execute consistently across Java platforms. However, this isn't always borne out in practice. For example, many Java platforms rely on the underlying operating system to schedule threads (discussed in Chapter 4), and the thread scheduling implementation varies from operating system to operating system. As a result, you must be careful to ensure that the program is designed to adapt to these vagaries.

The Java platform was also designed with security in mind. As well as the bytecode verifier, the platform provides a security framework to help ensure that malicious programs don't corrupt the underlying environment on which the program is running. Appendix C discusses Java's security framework.

## Installing and Working with JDK 7

Three software development kits (SDKs) exist for developing different kinds of Java programs:

- The Java SE (Standard Edition) Software Development Kit (known as the JDK) is used to create desktop-oriented *standalone applications* and web browser-embedded applications known as *applets*. You are introduced to standalone applications later in this section. I don't discuss applets because they aren't as popular as they once were.
- The Java ME (Mobile Edition) SDK is used to create applications known as MIDlets and Xlets. *MIDlets* target mobile devices, which have small graphical displays, simple numeric keypad interfaces, and limited HTTP-based network access. *Xlets* typically target television-oriented devices such as Blu-ray Disc players. The Java ME SDK requires that the JDK also be installed. I don't discuss MIDlets or Xlets.
- The Java EE (Enterprise Edition) SDK is used to create component-based enterprise applications. Components include *servlets*, which can be thought of as the server equivalent of applets, and servlet-based Java Server Pages (JSPs). The Java EE SDK requires that the JDK also be installed. I don't discuss servlets.

This section introduces you to JDK 7 (also referred to as *Java 7*, a term used in later chapters) by first showing you how to install this latest major Java SE release. It then shows you how to use JDK 7 tools to develop a simple standalone application—I'll use the shorter *application* term from now on.

### Installing JDK 7

Point your browser to `http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html` and follow the instructions on the resulting web page to download the appropriate JDK 7 installation exe or gzip tarball file for your Windows, Solaris, or Linux platform.

Following the download, run the Windows executable or unarchive the Solaris/Linux gzip tarball, and modify your `PATH` environment variable to include the resulting home directory's `bin` subdirectory so that you can run JDK 7 tools from anywhere in your filesystem. For example, you might include the `C:\Program Files\Java\jdk1.7.0` home directory in the `PATH` on a Windows platform. You should also update your `JAVA_HOME` environment variable to point to JDK 7's home directory, to ensure that any Java-dependent software can find this directory.

JDK 7's home directory contains several files (e.g., README.html and LICENSE) and subdirectories. The most important subdirectory from this book's perspective is bin, which contains various tools that we'll use throughout this book. The following list identifies some of these tools:

- jar: a tool for packaging classfiles and resource files into special ZIP files with ".jar" file extensions
- java: a tool for running applications
- javac: a tool that launches the Java compiler to compile one or more source files
- javadoc: a tool that generates special HTML-based documentation from Javadoc comments

The JDK's tools are run in a command-line environment. You establish this by launching a command window (Windows) or shell (Linux/Solaris), which presents to you a sequence of prompts for entering *commands* (program names and their arguments). For example, a command window (on Windows platforms) prompts you to enter a command by presenting a drive letter and path combination (e.g., C:\).

You respond to the prompt by typing the command, and then press the Return/Enter key to tell the operating system to execute the command. For example, javac x.java followed by a Return/Enter key press causes the operating system to launch the javac tool, and to pass the name of the source file being compiled (x.java) to this tool as its command-line argument. If you specified the asterisk (*) wildcard character, as in javac *.java, javac would compile all source files in the current directory. To learn more about working at the command line, check out Wikipedia's "Command-line interface" entry (http://en.wikipedia.org/wiki/Command-line_interface).

Another important subdirectory is jre, which stores the JDK's private copy of the Java Runtime Environment (JRE). The JRE implements the Java platform, making it possible to run Java programs. Users interested in running (but not developing) Java programs would download the public JRE. Because the JDK contains its own copy of the JRE, developers do not need to download and install the public JRE.

---

■ **Note** JDK 7 comes with external documentation that includes an extensive reference to Java's many *APIs* (see http://en.wikipedia.org/wiki/Application_programming_interface to learn about this term). You can download the documentation archive from http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html so that you can view this documentation offline. However, because the archive is fairly large, you might prefer to view the documentation online at http://download.oracle.com/javase/7/docs/index.html.

---

## Working with JDK 7

An application consists of a class with an entry-point method named main. Although a proper discussion of classes and methods must wait until Chapter 2, it suffices for now to just think of a class as a factory for creating objects (also discussed in Chapter 2), and to think of a method as a named sequence of instructions that are executed when the method is called. Listing 1-1 introduces you to your first application.

***Listing 1-1.*** *Greetings from Java*

```
class HelloWorld
{
   public static void main(String[] args)
   {
      System.out.println("Hello, world!");
   }
}
```

Listing 1-1 declares a class named `HelloWorld` that provides a framework for this simple application. It also declares a method named `main` within this class. When you run this application, and you will learn how to do so shortly, it is this entry-point method that is called and its instructions that are executed.

The `main()` method includes a header that identifies this method and a block of code located between an open brace character (`{`) and a close brace character (`}`). As well as naming this method, the header provides the following information:

- `public`: This reserved word makes `main()` visible to the startup code that calls this method. If `public` wasn't present, the compiler would output an error message stating that it could not find a `main()` method.
- `static`: This reserved word causes this method to associate with the class instead of associating with any objects created from this class. Because the startup code that calls `main()` doesn't create an object from the class in order to call this method, it requires that the method be declared `static`. Although the compiler will not report an error if `static` is missing, it will not be possible to run `HelloWorld`, which will not be an application if the proper `main()` method doesn't exist.
- `void`: This reserved word indicates that the method doesn't return a value. If you change `void` to a type's reserved word (e.g., `int`) and then insert a statement that returns a value of this type (e.g., `return 0;`), the compiler will not report an error. However, you won't be able to run `HelloWorld` because the proper `main()` method would not exist.
- `(String[] args)`: This parameter list consists of a single parameter named `args` of type `String[]`. Startup code passes a sequence of command-line arguments to `args`, which makes these arguments available to the code that executes within `main()`. You'll learn about parameters and arguments in Chapter 2.

The block of code consists of a single `System.out.println("Hello, world!");` method call. From left to write, `System` identifies a standard class of system utilities, `out` identifies an object variable located in `System` whose methods let you output values of various types optionally followed by a newline character to the standard output device, `println` identifies a method that prints its argument followed by a newline character to standard output, and `"Hello, world!"` is a *string* (a sequence of characters delimited by double quote `"` characters and treated as a unit) that is passed as the argument to `println` and written to standard output (the starting `"` and ending `"` double quote characters are not written; these characters delimit but are not part of the string).

■ **Note** All desktop Java/nonJava applications can be run at the command line. Before graphical user interfaces with their controls for inputting and outputting values (e.g., textfields), these applications obtained their input and generated their output with the help of *Standard I/O*, an input/output mechanism that originated with the Unix operating system, and which consists of standard input, standard output, and standard error devices.

The user would input data via the standard input device (typically the keyboard, but a file could be specified instead—Unix treats everything as files). The application's output would appear on the standard output device (typically a computer screen, but optionally a file or printer). Output messages denoting errors would be output to the standard error device (screen, file, or printer) so that these messages could be handled separately.

Now that you understand how Listing 1-1 works, you'll want to create this application. Complete the following steps to accomplish this task:

1. Copy Listing 1-1 to a file named `HelloWorld.java`.
2. Execute `javac HelloWorld.java` to compile this source file. `javac` will complain if you do not specify the "`.java`" file extension.

If all goes well, you should see a `HelloWorld.class` file in the current directory. Now execute `java HelloWorld` to run this classfile's `main()` method. Don't specify the "`.class`" file extension or `java` will complain. You should observe the following output:

```
Hello, world!
```

Congratulations! You have run your first Java-based application. You'll have an opportunity to run more applications throughout this book.

## Installing and Working with NetBeans 7

For small projects, it's no big deal to work at the command line with JDK tools. Because you'll probably find this scenario tedious (and even unworkable) for larger projects, you should consider obtaining an Integrated Development Environment (IDE) tool.

Three popular IDEs for Java development are Eclipse (`http://www.eclipse.org/`), IntelliJ IDEA (`http://www.jetbrains.com/idea/`), which is free to try but must be purchased if you want to continue to use it, and NetBeans (`http://netbeans.org/`). I focus on the NetBeans 7 IDE in this section because of its JDK 7 support. (IntelliJ IDEA 10.5 also supports JDK 7.)

■ **Note** For a list of NetBeans 7 IDE enhancements that are specific to JDK 7, check out the page at `http://wiki.netbeans.org/NewAndNoteworthyNB70#JDK7_support`.

This section shows you how to install the NetBeans 7 IDE. It then introduces you to this IDE while developing `HelloWorld`.

---

■ **Note** NetBeans is more than an IDE. It's also a platform framework that lets developers create applications much faster by leveraging the modular NetBeans architecture.

---

## Installing NetBeans 7

Point your browser to `http://netbeans.org/downloads/` and perform the following tasks:

1. Select an appropriate IDE language (English is the default).
2. Select an appropriate platform (Windows is the default).
3. Click the Download button underneath the next-to-leftmost (Java EE) column to initiate the download process for the appropriate installer file. I chose to download the English Java EE installer for the Windows platform, which is a file named netbeans-7.x-ml-javaee-windows.exe. (Because I don't explore Java EE in Beginning Java 7, it might seem pointless to install the Java EE version of NetBeans. However, you might as well install this software now in case you decide to explore Java EE after reading this book.)

Run the installer. After configuring itself, the installer presents a Welcome dialog that gives you the option of choosing which application servers you want to install with the IDE. Ensure that both the GlassFish Server and Apache Tomcat checkboxes remain checked (you might want to play with both application servers when exploring Java EE), and click the Next button.

On the resulting License Agreement dialog, read the agreement, indicate its acceptance by checking the checkbox, and click Next. Repeat this process on the subsequent JUnit License Agreement dialog.

The resulting NetBeans IDE 7.0 Installation dialog presents the default location where NetBeans will be installed (`C:\Program Files\NetBeans 7.0` on my platform) and the JDK 7 home directory location (`C:\Program Files\Java\jdk1.7.0` on my platform). Change these locations if necessary and click Next.

The resulting GlassFish 3.1 Installation dialog box presents the default location where the GlassFish application server will be installed (`C:\Program Files\glassfish-3.1` on my platform). Change this location if necessary and click Next.

The resulting Apache Tomcat 7.0.11 Installation dialog presents the default location where the Apache Tomcat application server will be installed (`C:\Program Files\Apache Software Foundation\Apache Tomcat 7.0.11` on my platform). Change this location if necessary and click Next.

The resulting Summary dialog presents your chosen options as well as the combined installation size for all software being installed. After reviewing this information, click the Install button to begin installation.

Installation takes a few minutes and culminates in a Setup Complete dialog. After reviewing this dialog's information, click the Finish button to complete installation.

Assuming a successful installation, start this IDE. NetBeans first presents a splash screen while it performs various initialization tasks, and then presents a main window similar to that shown in Figure 1-1.

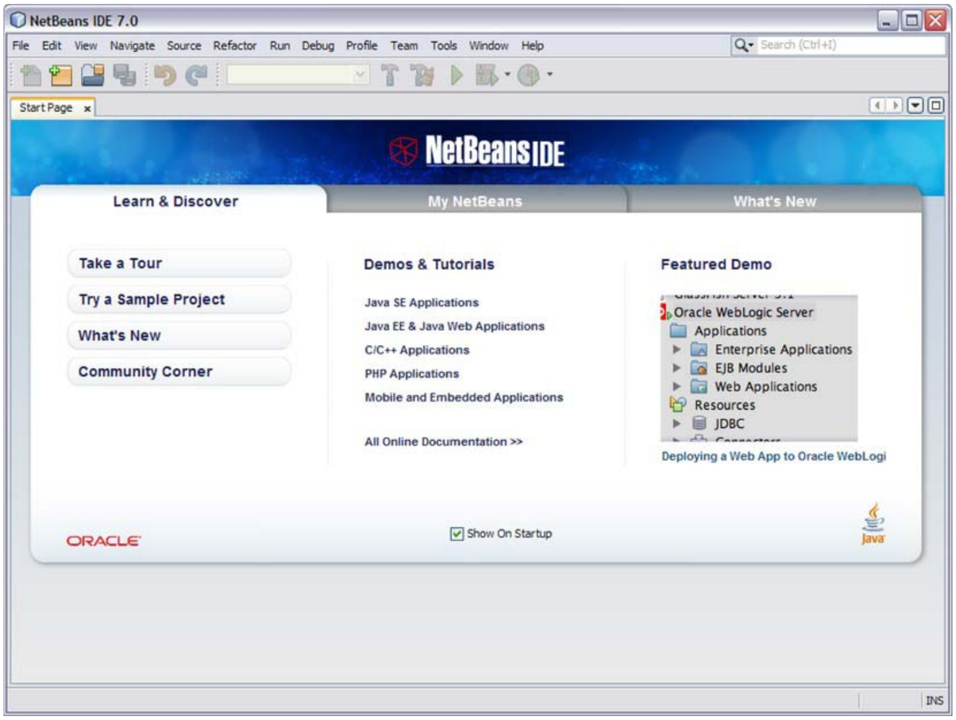

***Figure 1-1.** The NetBeans 7 IDE's main window initially presents a Start Page tab.*

If you've worked with previous versions of the NetBeans IDE, you might want to click the Take a Tour button to learn how version 7 differs from its predecessors. You are taken to a web page that provides video tours of the IDE, such as NetBeans IDE 7.0 Overview.

## Working with NetBeans 7

NetBeans presents a user interface whose main window is divided into a menu bar, a toolbar, a workspace, and a status bar. The workspace presents a Start Page tab for learning about NetBeans, accessing your NetBeans projects, and more.

To help you get comfortable with this IDE, I'll show you how to create a `HelloWorld` project that reuses Listing 1-1's source code. I'll also show you how to compile and run the `HelloWorld` application. Complete the following steps to create the `HelloWorld` project:

1. Select New Project from the File menu.
2. Make sure that Java is the selected category and Java Application is the selected Project in their respective Categories and Projects lists on the resulting New Project dialog box's Choose Project pane. Click Next.
3. On the resulting Name and Location pane, enter **`HelloWorld`** into the Project Name textfield. Notice that `helloworld.HelloWorld` appears in the textfield to the right of the Create Main Class checkbox (which must be checked). The