Learn Java Fundamentals

A Primer for Java Development and Programming

Jeff Friesen

Apress®

Learn Java Fundamentals

A Primer for Java Development and Programming

Jeff Friesen

Apress®

Learn Java Fundamentals: A Primer for Java Development and Programming

Jeff Friesen
Dauphin, MB, Canada

ISBN-13 (pbk): 979-8-8688-0350-5 ISBN-13 (electronic): 979-8-8688-0351-2
https://doi.org/10.1007/979-8-8688-0351-2

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: James Robinson-Prior
Development Editor: James Markham
Coordinating Editor: Gryffin Winkler

Cover designed by eStudioCalamar

Photo by pariwat pannium on Unsplash

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles 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 Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

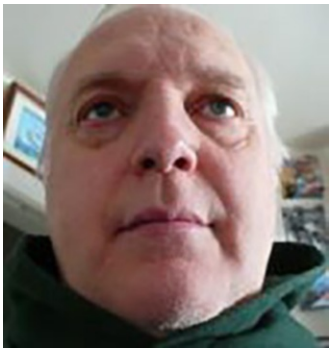
Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress). For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

To my Lord and Savior, Jesus Christ
and
To the memories of my parents and my older sister
and
To my younger sister and her family.

Table of Contents

About the Author



Jeff Friesen is a freelance teacher and software developer with an emphasis on Java. In addition to authoring several books on Java and Android for Apress such as *Java I/O, NIO and NIO.2* and *Java XML and JSON*, Jeff has written numerous articles on Java and other technologies for JavaWorld (now known as InfoWorld), InformIT, Java.net, SitePoint, and other websites.

About the Technical Reviewer

Massimo Nardone is a seasoned cyber, information, and operational technology (OT) security professional with 28 years of experience working with companies such as IBM, HP, and Cognizant, with IT, OT, IoT, and IIoT security roles and responsibilities including CISO, BISO, IT/OT/IoT Security Architect, Security Assessor/Auditor, PCI QSA, and ICS/SCADA Expert. He is the founder of Massimo Security Services, a company providing IT-OT-IoT security consulting services, and member of ISACA, ISF, Nordic CISO Forum, and Android Global Forum and owns four international patents. He is coauthor of five Apress IT books.

Introduction

Java is a popular programming language and environment. Because it is used in the Information Technology departments of many companies, learning Java is a great way to boost your career (and earn more money in these difficult financial times).

If you have never worked with Java, this 14-chapter book is for you. Chapter 1 starts you on a gentle journey to learn Java fundamentals.

Chapters 2 through 11 focus mainly on language syntax, although a few APIs that are closely related to syntax are also presented.

Chapter 2 focuses on comments, identifiers, types, variables, and literals. These features are fundamental to many languages, and this chapter also shows you where Java differs from other languages in their implementation.

Chapter 3 focuses on expressions (and operators), and Chapter 4 focuses on statements. Again, these features are found in many languages. You will use these building blocks to construct simple Java programs and will learn where Java's implementations of expressions (and operators) and statements diverge from other languages.

Chapter 5 focuses on arrays. You will use this fundamental data structure to create programs that work with sequences of data items. For example, you might want to search a sequence of employee IDs for a specific identifier.

If this was all that Java had to offer, you would be able to create sophisticated structured programs. In a structured program, data and operations that manipulate the data are separated. However, Java goes beyond its ability to create structured programs, as revealed in Chapters 6 through 8.

Chapter 6 introduces you to classes and objects. A class is a *template* from which objects are manufactured. It provides an architecture for structuring data and associating that data with code that manipulates the data. An *object* is an instance of a class (kind of like a cookie is an instance of a cookie cutter). It stores data that can be manipulated by the code that the object receives from its class. (Don't worry if this seems complicated. After reading Chapter 6, you'll have a much better understanding.)

Java and other languages that support classes and objects are known as *object-based languages*. To go beyond object based and become an *object-oriented language*,

a language must also support inheritance. Java supports inheritance, which you'll learn about in Chapter 7.

Programs can be made more efficient through polymorphism, which is based on inheritance. The idea behind polymorphism is that a single symbol can represent many different types (e.g., the + symbol can represent integer addition, floating-point addition, or string concatenation). You'll learn about polymorphism in Chapter 8.

There are a few more language features that you need to learn about before you can tour Java's many APIs. Chapter 9 begins by introducing you to static, non-static, local, and anonymous classes. These features let you logically organize your code, making it more readable and maintainable.

Packages let you organize related classes in the equivalent of a file folder. This feature helps you avoid name conflicts by organizing a library of classes under a single prefix. Check out Chapter 10 to learn about packages.

Java provides a robust exception-handling mechanism for dealing with flawed code or unexpected difficulties, such as attempting to open a nonexistent file. This mechanism is covered in Chapter 11.

The final three chapters tour some fundamental APIs that you'll use in many Java programs. Chapter 12 focuses on the `Math` class and related types, Chapter 13 focuses on `String` and `StringBuffer`, and Chapter 14 focuses on `System`. After you explore these types, you'll be able to explore additional APIs on your own to increase your Java knowledge.

Two appendixes round out this book. Appendix A provides a quick reference to Java's supported reserved words, and Appendix B provides a quick reference to Java's supported operators.

CHAPTER 1

Getting Started with Java

Welcome to Java. This technology is widely used in the business world, and you probably want to learn it quickly so you can capture a job in one of these companies as a Java programmer. Although Java is vast and constantly evolving, there are various fundamental features that are timeless and easy to understand. After you master these fundamentals, you will have an easier time writing Java programs.

This chapter launches you on a tour of Java's fundamental features. You first receive an answer to the "What is Java?" question. Next, you learn about the Java Development Kit, which is the necessary software for developing Java programs on your computer. Moving on, you are introduced to your first Java program, which outputs a simple "hello, world" message. Finally, you discover application architecture.

Note A *program* is a sequence of instructions for a computer to execute. An *application* is a program with a single entry point of execution. (In contrast, an *applet* – an old form of Java program that is no longer widely used – has multiple entry points.) For example, a Microsoft Windows program that is stored in an `.exe` file has a single entry point. When expressed in C language *source code* (textual instructions), the entry point is defined by a *function* (a named sequence of instructions) with the name `main`.

What Is Java?

Java is like a two-sided coin. From one side, it's a computer programming language. Conversely, it's a virtual *platform* (the hardware and software context in which a program runs) for running programs written in that language.

J. Friesen, *Learn Java Fundamentals*, https://doi.org/10.1007/979-8-8688-0351-2_1

Note Java has an interesting 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 to learn more.

Java Is a Programming Language

Java is a programming language with many features that are identical to those found in the C and C++ languages. This is no accident. One of Java's initial goals was to make it easy for C/C++ programmers to migrate to Java to quickly build up an initial pool of programmers that would help Java become successful.

You will discover several similarities between these languages:

- The same single-line and multiline comments for documenting source code are found in Java and C/C++.
- Various identical reserved words are found in Java and C/C++, such as if, while, for, and switch. Various other reserved words are found in Java and C++ but not in C, such as `try`, `catch`, `class`, and `public`.
- Primitive types are shared between the three languages: character and integer are examples. Furthermore, reserved words for these types are shared between these languages: char and int are examples.
- Many of the same operators are shared between Java and C/C++. Arithmetic operators (such as * and +) and relational operators (such as == and <=) are examples.
- Finally, Java and C/C++ use brace characters ({ and }) to delimit blocks of statements.

Java also differs from C/C++ in many ways. Here are a few of the many differences:

- Java supports an additional comment style for documenting source code. This comment style is known as *Javadoc*.
- Java provides reserved words that are not found in C/C++. Examples include `strictfp` and `transient`.

- Java's character type is larger than the character type in C and C++. In those languages, a character occupies one byte of memory. In contrast, Java's character type occupies two bytes.
- Java doesn't support all of C/C++'s operators. For example, you won't find the C/C++ `sizeof` operator in Java. Also, the `>>>` (unsigned right shift) operator is exclusive to Java.
- Java provides labeled `break` and `continue` statements. These variants of their C/C++ counterparts, which don't accept labels, are a safer alternative to C/C++'s `goto` statement, which Java doesn't support.

I discuss comments, reserved words, types, operators, and statements later in this book.

The Java programming language is rigorously defined by various rules that describe its *syntax* (structure) and *semantics* (meaning). These rules are used by a compiler to verify correctness when translating a program's source code into equivalent *bytecode*, which is a portable representation of the program's executable code. This bytecode is stored in one or more *class files*, which are the Java equivalent of a Windows program's executable (`.exe`) file.

Java Is a Virtual Platform

Java is a virtual platform that executes Java programs. Unlike real platforms that consist of a microprocessor (such as an Intel or AMD processor) and operating system (such as Windows 11), the Java platform consists of virtual machine and execution environment software.

A *virtual machine* is a software-based processor with its own set of instructions. The Java Virtual Machine's (JVM) associated *execution environment* consists of a huge library of prebuilt reference types (think Application Program Interfaces [APIs]) that Java programs can use to perform routine tasks (such as opening a file). The execution environment also contains "glue" code that connects the JVM to the underlying operating system via the Java Native Interface. (I don't discuss the Java Native Interface in this book because I don't consider it to be a fundamental feature.)

Note The combination of bytecode and the virtual machine makes it possible to achieve *portability*: the same Java program runs on all platforms that support the virtual machine. It's not necessary to recompile the program's source code for each platform.

A Java program is run by a special executable, which I call the *program launcher*. Because a program consists of one or more class files, the launcher receives the name of the *main class file* (the class file where execution begins). After loading the JVM into memory, it tells the JVM to use its *class loader* component to load the main class file into memory. The JVM then verifies that the class file's bytecode is safe to run (e.g., no viruses) and runs it.

Note The verifier and a security manager architecture make it possible to achieve *security*: a Java application will not be allowed to run when the verifier detects corrupt bytecode. Furthermore, when a security manager is installed, the application won't be able to steal sensitive information, erase files, or otherwise harm a user's computer.

During execution, a class file might refer to another class file. When this happens, the JVM uses the class loader to load the referenced class file into memory and then verifies and (if okay to run) executes that class file's bytecode.

The Java Development Kit

The Java Development Kit (JDK) provides the necessary software for creating *Java applications*, which are a category of Java programs with a single entry point of execution. They contrast with *Java applets*, another category of Java programs that run embedded in web pages. Applets are rarely used these days.

Follow these steps to download the JDK:

1. Enter `www.oracle.com/java/technologies/` in your browser. This takes you to the main page of Oracle's Java site.

2. At the time of writing, the newest download is version 21.0.1. Click on the **Java SE 21.0.1** link. (Java SE stands for Java Standard Edition. This is the foundation on which other editions are based and is the appropriate edition for this book. Another edition is Java EE, for Java Enterprise Edition. You would use this edition when developing complex business solutions involving web servers, database management systems, and client computers.)

3. In the **JDK Development Kit 21.0.1 downloads** section of the resulting **Java Downloads** page, you will see **Linux**, **macOS**, and **Windows** tabs. The JDK is available for all three operating systems. Choose whichever one is right for you. For example, I clicked the **Windows** tab because I was running Windows. I then had a choice between different kinds of installers. I chose the x64 installer whose file name ends with a `.exe` file extension. I found this the easiest way to install the JDK.

Once you download the installer, such as `jdk-21_windows-x64_bin.exe`, run this program and follow the onscreen prompts to install the JDK.

The JDK contains various tools for use in application development. Four of these tools are the Java compiler (`javac.exe` in the Windows download), the Java program launcher (`java.exe` in the Windows download), the Java documentation generator (`javadoc.exe` in the Windows download), and the Java archiver (`jar.exe` in the Windows download). You will only need to work with these tools in this book.

The JDK's compiler, program launcher, documentation generator, archiver, and other tools are designed to be run from the command line within the context of a *console* (an operating system–specific construct consisting of a window for viewing output and a command line for obtaining command-based input). To obtain a console on Windows operating systems, perform the following tasks:

1. Go to the **Start** menu and select **Run**.

2. In the **Run** dialog box, enter **cmd** in the text field and click the **OK** button. On the Windows operating system, you should observe a window similar to that shown in Figure 1-1.

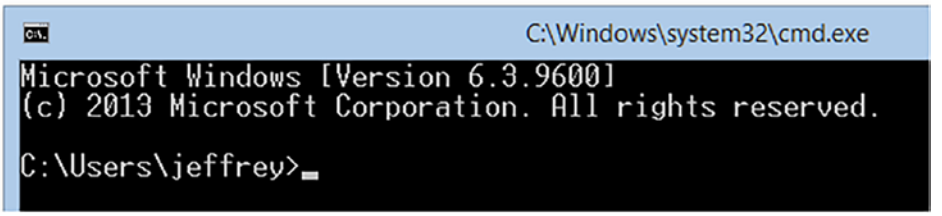


Figure 1-1. *The upper portion of a console as seen on a Windows 8.1 machine*

Figure 1-1 reveals **C:\Users\jeffrey>**, which is a prompt for entering a command on my Windows 8.1 machine. The rectangular box to the right of > is the *cursor*, which indicates the current position for entering text on the command line.

"hello, world" – Java Style

Let's create a simple application to get a taste of Java code. Traditionally, the first application does nothing but output the message `hello, world` on the console. Listing 1-1 presents the source code to a `HelloWorld` application that does just that.

Listing 1-1. `HelloWorld.java`

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

Listing 1-1 declares a `HelloWorld` class (I explain classes in Chapter 6) that serves as a placeholder for the `main()` *method* (a named sequence of instructions that executes in the context of a class).

Note Languages such as C use functions instead of methods. A *function* is a named sequence of instructions that executes outside of any context.

The `main()` method serves as the entry point to the application. When the application runs, `main()`'s code is executed.

The `main()` method header (`public static void main(String[] args)`) exhibits some interesting features:

- The method is marked `public` so that the `Java` program launcher can locate it. If `public` is absent, an error message is output when attempting to run the application.
- The method is marked `static` so that a `HelloWorld` object does not need to be created in order to call `main()`. The launcher calls `main()` directly. It knows nothing of objects. If `static` is absent, an error message is output when attempting to run the application.
- The method is declared with a parameter list consisting of `String[] args`, which identifies an array of string arguments that are passed after the application's name (`HelloWorld`) on the command line when the application is run by the launcher. A *string is* a sequence of characters placed between double quotes (").
- The method is declared with a `void` return type that signifies the method returns nothing.

Don't worry if concepts such as return type and parameter list are confusing. You'll learn about these concepts later in this book.

The `main()` method executes `System.out.println("hello, world");` to output `hello, world` on the console's window. I explore `System.out` and its `System.err` and `System.in` counterparts in Chapter 14.

INDENTATION, OPEN BRACE CHARACTER PLACEMENT, AND CODE-SEPARATION STYLES

Programmers often follow one style when indenting source code, another style when positioning a block's open brace character, and a third style when using blank lines to separate segments of source code. (I briefly discuss *blocks*, which are sequences of code surrounded by { and } characters, in Chapter 4.)

Listing 1-1 demonstrates the first two style categories. It shows my tendency to indent, by three spaces, all lines in a block. I find that doing so makes it easier to follow the organization of my source code when updating it as requirements change.

Also, Listing 1-1 shows my tendency to align the open ({) and close (}) brace characters, so I can more easily locate the start and end of a block. Many programmers prefer the following brace character alignment instead:

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

Another style issue involves inserting blank lines to separate segments of code, where each segment consists of statements that work collectively on some aspect of the program. Here is a contrived example, involving a pair of classes, A and B:

```
class A
{
   void method1()
   {
      for (int i = 0; i < 10; i++)
         System.out.println(i);

      while (true)
      {
         // ... do something here
      }
   }

   void method2()
   {
      for (int i = 0; i < 10; i++)
         System.out.println(i);

      while (true)
      {
         // ... do something here
      }
   }
}
```

```
class B
{
   void method1()
   {
      for (int i = 0; i < 10; i++)
         System.out.println(i);
      while (true)
      {
         // ... do something here
      }
   }
   void method2()
   {
      for (int i = 0; i < 10; i++)
         System.out.println(i);
      while (true)
      {
         // ... do something here
      }
   }
}
```

Each of classes A and B declares two methods: `method1()` and `method2()`. Furthermore, each of `method1()` and `method2()` declares a `for` statement followed by a `while` statement.

Don't worry about classes, methods, and statements. I cover classes and methods in Chapter 6 and cover statements in Chapter 4.

For now, pay attention to the blank line styles in each of A and B. A's style is to place a blank line between each method and between each group of related statements. B's style is to eliminate the blank line from between the methods and from between the statements.

Form your own styles for indentation, brace character placement, and code separation. Although these styles don't impact the generated code, adhering to them religiously sets you apart from other programmers and can make your source code easier to read and maintain. I tend to vary my code-separation style, which you'll discover throughout this book's code listings.

Compile the source code as follows (you must include the `.java` file extension):

```
javac HelloWorld.java
```

If everything goes well, you should observe a `HelloWorld.class` file in the current directory.

Now, execute the following command to run `HelloWorld.class` (you must not include the `.class` file extension):

```
java HelloWorld
```

If all goes well, you should observe the following output:

```
hello, world
```

Congratulations! You've just run your first Java application. You should feel proud.

Application Architecture

An application consists of at least one class, and this class must declare a `main()` entry-point method, as you saw in Listing 1-1. However, many applications will consist of multiple classes. All of these classes might be declared in a single source file, or each class might be declared in its own source file. Consider Listing 1-2.

Listing 1-2. `Classes.java`

```
class A
{
   static void a()
   {
       System.out.println("a() called");
   }
}

class B
{
   static void b()
```

```
   {
       System.out.println("b() called");
   }
}

class C
{
   public static void main(String[] args)
   {
      A.a();
      B.b();
   }
}
```

Listing 1-2 declares three classes (`A`, `B`, and `C`) in the same source file – `Classes.java`. Class `C` is the entry-point class because it declares the `main()` method.

Compile `Classes.java` as follows:

```
javac Classes.java
```

You should observe `A.class`, `B.class`, and `C.class` class files in the current directory.

Run this application as follows:

```
java C
```

You should observe the following output:

```
a() called
b() called
```

If you try to execute `A` (`java A`) or `B` (`java B`), you'll discover an error message because neither class declares the `main()` entry-point method.

This brings up an interesting point. You could declare `main()` methods in `A` and `B` and run these classes as applications. However, this could get confusing.

You might want to declare a `main()` method in each of `A` and `B` to test these classes, but there's probably no other good reason to do so. It's best to avoid confusion by declaring `main()` in the entry-point class only.

What's Next?

Now that you've had a taste of Java, it's time to build on that knowledge by exploring language features. Chapter 2 begins this process by focusing on the most basic language features: comments, identifiers (and reserved words), types, variables, and literals.

CHAPTER 2

Comments, Identifiers, Types, Variables, and Literals

When learning a new programming language, starting with the most basic of language features is best. These features are comments, identifiers (with reserved words as a subset), types, variables, and literals. This chapter introduces you to these features in a Java context.

Comments

It's important to document your source code so that you and anyone else who might maintain it in the future can understand the code's purpose. Our brains tend to forget things as we age, and we may not understand why we wrote the code the way we did. Source code should be documented when it is written. This documentation might have to be modified whenever the code is changed so that it accurately explains the new code.

Java provides *comments* for documenting source code. Whenever you compile the source code, the compiler ignores the comments – no bytecode is generated. Single-line, multiline, and Javadoc (documentation) comments are supported.

J. Friesen, *Learn Java Fundamentals*, https://doi.org/10.1007/979-8-8688-0351-2_2

Single-Line Comments

A *single-line comment* appears on one line of source code. It begins with the `//` character sequence and continues to the end of the line. The compiler ignores everything on this line starting with the `//` characters. The following example demonstrates a single-line comment:

```
double degrees = (5.0 / 9.0) * (x - 32.0); // Convert x degrees Fahrenheit
to Celsius.
```

Single-line comments are useful for specifying short but meaningful information. They shouldn't be used to insert unhelpful information, for example, `// This is a comment`.

Multiline Comments

A *multiline comment* typically extends over multiple lines of source code although it can appear on a single line. This comment begins with `/*` and ends with `*/`. The compiler ignores everything in between (including `/*` and `*/`). The following example demonstrates a multiline comment:

```
/* Extract both components of an email address into a two-element array.
   email_parts[0] stores "xyz" and email_parts[1] stores "gmail.com". */
String[] email_parts = "xyz@gmail.com".split("@", 2); // extract
```

You cannot nest a multiline comment inside of another multiline comment. For example, the compiler generates an error when it encounters the following nested comments:

```
/*
  /*
    Nested multiline comments are illegal.
  */
*/
```

Caution The compiler reports an error when it encounters nested multiline comments.

Javadoc Comments

A *Javadoc comment* is a variation of the multiline comment. It begins with /** (instead of /*) and (like a multiline comment) ends with */. All characters from /** through */ are ignored by the compiler. The following example presents a Javadoc comment:

```
/**
 *  Application entry point
 *
 *  @param args array of command-line arguments passed to this method
 */
public static void main(String[] args)
{
   // TODO code application logic here
}
```

This example's Javadoc comment describes an application's `main()` method. Sandwiched between /** and */ is a description of the method and the `@param` *Javadoc tag* (an `@`-prefixed instruction to the `javadoc` tool).

Here is a list of some commonly used Javadoc tags (including `@param`):

- `@author` identifies the source code's author.
- `@deprecated` identifies a source code entity (such as a method) that should no longer be used.
- `@param` identifies one of a method's parameters.
- `@see` provides a see-also reference.
- `@since` identifies the software release where the entity first originated.
- `@return` identifies the kind of value that the method returns.
- `@throws` documents an exception thrown from a method.

Listing 2-1 presents updated source code to Listing 1-1's `HelloWorld` application. This source code includes a pair of Javadoc comments that document the `HelloWorld` class and its `main()` entry-point method.