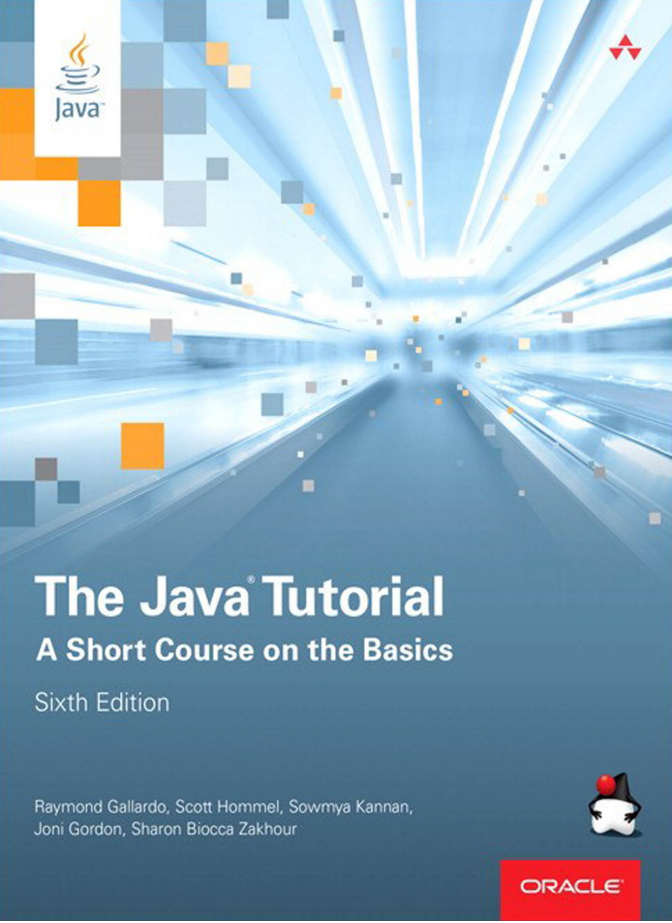
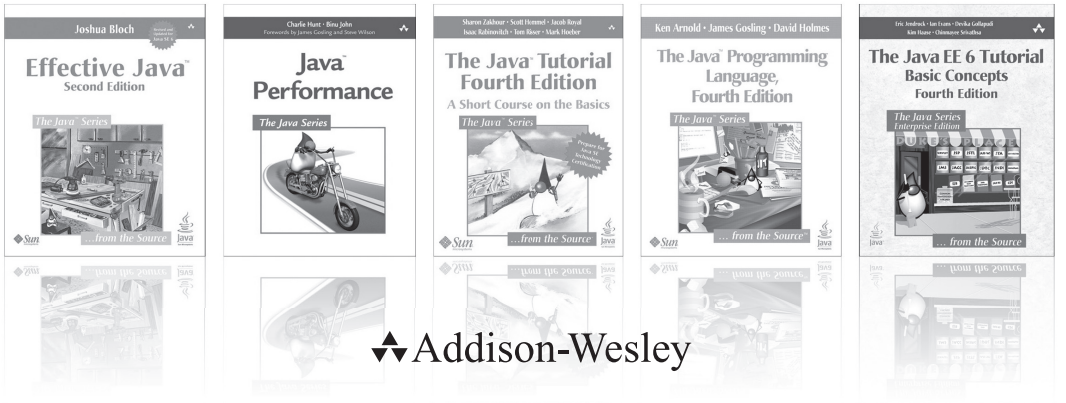


The Java® Tutorial

Sixth Edition

The Java® Series



Visit informit.com/thejavaseries for a complete list of available publications.

Since 1996, when Addison-Wesley published the first edition of *The Java Programming Language* by Ken Arnold and James Gosling, this series has been the place to go for complete, expert, and definitive information on Java technology. The books in this series provide the detailed information developers need to build effective, robust, and portable applications and are an indispensable resource for anyone using the Java platform.

Make sure to connect with us!
informit.com/socialconnect

Addison Wesley | informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE | Safari Books Online

The Java® Tutorial

A Short Course on the Basics

Sixth Edition

Raymond Gallardo
Scott Hommel
Sowmya Kannan
Joni Gordon
Sharon Biocca Zakhour

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data
Gallardo, Raymond.
The Java tutorial : a short course on the basics / Raymond Gallardo, Scott Hommel, Sowmya Kannan, Joni Gordon, Sharon Biocca Zakhour.—Sixth edition.
pages cm
Previous edition: The Java tutorial : a short course on the basics / Sharon Zakhour, Sowmya Kannan, Raymond Gallardo. 2013, which was originally based on The Java tutorial / by Mary Campione.
Includes index.
ISBN 978-0-13-403408-9 (pbk. : alk. paper)—ISBN 0-13-403408-2 (pbk. : alk. paper)
1. Java (Computer program language) I. Title.
QA76.73.J38Z35 2015
005.13'3—dc23

2014035811

ISBN-13: 978-0-13-403408-9
ISBN-10: 0-13-403408-2

Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
First printing, December 2014

Contents

This page intentionally left blank

Preface

Since the acquisition of Sun Microsystems by Oracle Corporation in early 2010, it has been an exciting time for the Java language. As evidenced by the activities of the Java Community Process program, the Java language continues to evolve. The publication of this sixth edition of *The Java® Tutorial* reflects version 8 of the Java Platform Standard Edition (Java SE) and references the Application Programming Interface (API) of that release.

This edition introduces new features added to the platform since the publication of the fifth edition (under release 7):

- Lambda expressions enable you to treat functionality as a method argument or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly. See the new section in Chapter 4, "Lambda Expressions."
- Type annotations can be used in conjunction with pluggable type systems for improved type checking, and repeating annotations enable the application of the same annotation to a declaration or type use. See the new sections in Chapter 5, "Type Annotations and Pluggable Type Systems" and "Repeating Annotations."
- Default methods are methods in an interface that have an implementation. They enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces. See the new section in Chapter 6, "Default Methods."

- Aggregate operations enable you to perform functional-style operations on streams of elements—in particular, bulk operations on collections such as sequential or parallel map-reduce transformations. See the new section in Chapter 12, "Aggregate Operations."
- Improvements have been added that focus on limiting attackers from using malicious applets and rich Internet applications (RIAs). See the following new and updated sections:
  - Chapter 16, "Packaging Programs in JAR Files"
  - Chapter 19, "Security in Rich Internet Applications" and "Guidelines for Securing Rich Internet Applications"
  - Chapter 20, "Deployment Best Practices"
- Date-Time APIs enable you to represent dates and times and manipulate date and time values. They support the International Organization for Standardization (ISO) calendar system as well as other commonly used global calendars. See the new Chapter 21.

If you plan to take one of the Java SE 8 certification exams, this book can help. The appendix, "Preparation for Java Programming Language Certification," lists the three exams that are available, detailing the items covered by each exam, cross-referenced to places in the book where you can find more information about each topic. Note that this is one source, among others, that you will want to use to prepare for your exam. Check the online tutorial for the latest certification objectives and cross-references to sections of the tutorial.

All of the material has been thoroughly reviewed by members of Oracle Java engineering to ensure that the information is accurate and up to date. This book is based on the online tutorial hosted on Oracle Corporation's web site at the following URL:

```
http://docs.oracle.com/javase/tutorial/
```

The information in this book, often referred to as "the core tutorial," is required by most beginning to intermediate programmers. Once you have mastered this material, you can explore the rest of the Java platform documentation on the web site. If you are interested in developing sophisticated RIAs, check out JavaFX, the Java graphical user interface (GUI) toolkit, which comes with the Java SE Development Kit (JDK). To learn more, see Chapter 22, "Introduction to JavaFX."

As always, our goal is to create an easy-to-read, practical programmers' guide to help you learn how to use the rich environment provided by Java to build applications, applets, and components. Go forth and program!

Who Should Read This Book?

This book is geared toward both novice and experienced programmers:

- *New programmers* can benefit most from reading the book from beginning to end, including the step-by-step instructions for compiling and running your first program in Chapter 1, "Getting Started."
- *Programmers experienced with procedural languages* such as C may want to start with the material on object-oriented concepts and features of the Java programming language.
- *Experienced programmers* may want to jump feet first into the more advanced topics, such as generics, concurrency, or deployment.

This book contains information to address the learning needs of programmers with various levels of experience.

How to Use This Book

This book is designed so you can read it straight through or skip around from topic to topic. The information is presented in a logical order, and forward references are avoided wherever possible.

The examples in this book are compiled against the JDK 8 release. *You need to download this release (or later) in order to compile and run most examples*.

Some material referenced in this book is available online—for example, the downloadable examples, the solutions to the questions and exercises, the JDK 8 guides, and the API specification.

You will see footnotes like the following:

```
8/docs/api/java/lang/Class.html
```

and

```
tutorial/java/generics/examples/BoxDemo.java
```

The Java documentation home on the Oracle web site is at the following location:

```
http://docs.oracle.com/javase/
```

To locate the footnoted files online, prepend the URL for the Java documentation home:

```
http://docs.oracle.com/javase/8/docs/api/java/lang/Class.html
```

```
http://docs.oracle.com/javase/tutorial/java/generics/examples/BoxDemo.java
```

The Java Tutorials are also available in two eBook formats:

- mobi eBook files for Kindle
- ePub eBook files for iPad, Nook, and other devices that support the ePub format

Each eBook contains a single trail that is equivalent to several related chapters in this book. You can download the eBooks via the link "In Book Form" on the home page for the Java Tutorials:

```
http://docs.oracle.com/javase/tutorial/index.html
```

We welcome feedback on this edition. To contact us, please see the tutorial feedback page:

```
http://docs.oracle.com/javase/feedback.html
```

Acknowledgments

This book would not be what it is without the Oracle Java engineering team who tirelessly reviews the technical content of our writing. For this edition of the book, we especially want to thank Alan Bateman, Alex Buckley, Stephen Colebourne, Joe Darcy, Jeff Dinkins, Mike Duigou, Brian Goetz, Andy Herrick, Stuart Marks, Thomas Ng, Roger Riggs, Leif Samuelsson, and Daniel Smith.

Illustrators Jordan Douglas and Dawn Tyler created our professional graphics quickly and efficiently.

Editors Janet Blowney, Deborah Owens, and Susan Shepard provided careful and thorough copyedits of our JDK 8 work.

Thanks for the support of our team: Devika Gollapudi, Ram Goyal, and Alexey Zhebel.

Last but not least, thanks for the support of our management: Sowmya Kannan, Sophia Mikulinsky, Alan Sommerer, and Barbara Ramsey.

About the Authors

Raymond Gallardo is a senior technical writer at Oracle Corporation. His previous engagements include college instructor, technical writer for IBM, and bicycle courier. He obtained his BSc in computer science and English from the University of Toronto and MA in creative writing from the City College of New York.

Scott Hommel is a senior technical writer at Oracle Corporation, where he documents Java SE. For the past fifteen years, he has written tutorials, technical articles, and core release documentation for Java SE and related technologies.

Sowmya Kannan wears many hats on the Java SE documentation team, including planning, writing, communicating with developer audiences, and tinkering with production tools. She has more than fifteen years of experience in the design, development, and documentation of the Java platform, Java-based middleware, and web applications.

Joni Gordon is a principal technical writer at Oracle Corporation. She has contributed to the documentation for Java SE and JavaFX. She has been a technical writer for more than fifteen years and has a background in enterprise application development.

Sharon Biocca Zakhour was previously a principal technical writer on staff at Oracle Corporation and formerly at Sun Microsystems. She has contributed to Java SE documentation for more than twelve years, including *The Java™ Tutorial, Fourth Edition,* and *The JFC Swing Tutorial, Second Edition*. She graduated from UC Berkeley with a BA in computer science and has worked as a programmer, developer support engineer, and technical writer for thirty years.

This page intentionally left blank

This page intentionally left blank

3

Language Basics

Chapter Contents

You've already learned that objects store their state in fields. However, the Java programming language uses the term *variable* as well. The first section of this chapter discusses this relationship, plus variable naming rules and conventions, basic data types (primitive types, character strings, and arrays), default values, and literals.

The second section describes the operators of the Java programming language. It presents the most commonly used operators first and the less commonly used operators last. Each discussion includes code samples that you can compile and run.

Operators may be used for building expressions, which compute values; expressions are the core components of statements, and statements may be grouped into blocks. The third section discusses expressions, statements, and blocks using example code that you've already seen.

The final section describes the control flow statements supported by the Java programming language. It covers the decision-making, looping, and branching statements that enable your programs to conditionally execute particular blocks of code.

Note that each section contains its own questions and exercises to test your understanding.

Variables

As you learned in the previous chapter, an object stores its state in *fields*:


```
int cadence = 0;
int speed = 0;
int gear = 1;
```

In Chapter 2, the section "What Is an Object?" introduced you to fields, but you probably have still a few questions, such as, What are the rules and conventions for naming a field? Besides `int`, what other data types are there? Do fields have to be initialized when they are declared? Are fields assigned a default value if they are not explicitly initialized? We'll explore the answers to such questions in this chapter, but before we do, there are a few technical distinctions you must first become aware of. In the Java programming language, the terms *field* and *variable* are both used; this is a common source of confusion among new developers because both often seem to refer to the same thing. The Java programming language defines the following kinds of variables:

- *Instance variables (nonstatic fields).* Technically speaking, objects store their individual states in "nonstatic fields"—that is, fields declared without the `static` keyword. Nonstatic fields are also known as *instance variables* because their values are unique to each *instance* of a class (to each object, in other words); for example, the `currentSpeed` of one bicycle is independent of the `currentSpeed` of another.
- *Class variables (static fields).* A *class variable* is any field declared with the `static` modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. For example, a field defining the number of gears for a particular kind of bicycle could be marked as `static` since conceptually the same number of gears will apply to all instances. The code `static int numGears = 6;` would create such a static field. Additionally, the keyword `final` could be added to indicate that the number of gears will never change.
- *Local variables.* Similar to how an object stores its state in fields, a method will often store its temporary state in *local variables*. The syntax for declaring a local variable is similar to declaring a field (e.g., `int count = 0;`). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared—which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.

- *Parameters.* You've already seen examples of parameters, both in the `Bicycle` class and in the `main` method of the "Hello World!" application. Recall that the signature for the `main` method is `public static void main(String[] args)`. Here, the `args` variable is the parameter to this method. The important thing to remember is that parameters are always classified as *variables*, not *fields*. This applies to other parameter-accepting constructs as well (such as constructors and exception handlers) that you'll learn about later in the chapter.

That said, the remainder of the chapters use the following general guidelines when discussing fields and variables. If we are talking about *fields in general* (excluding local variables and parameters), we may simply use the term *fields*. If the discussion applies to *all of the above*, we may simply use the term *variables*. If the context calls for a distinction, we will use specific terms (such as *static field* or *local variables*) as appropriate. You may also occasionally see the term *member* used as well. A type's fields, methods, and nested types are collectively called its *members*.

Naming

Every programming language has its own set of rules and conventions for the kinds of names that you're allowed to use, and the Java programming language is no different. The rules and conventions for naming your variables can be summarized as follows:

- Variable names are case sensitive. A variable's name can be any legal identifier—an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign (`$`), or the underscore character (`_`). The convention, however, is to always begin your variable names with a letter, not `$` or `_`. Additionally, the dollar sign character, by convention, is never used at all. You may find some situations where autogenerated names will contain the dollar sign, but your variable names should always avoid using it. A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with `_`, this practice is discouraged. White space is not permitted.
- Subsequent characters may be letters, digits, dollar signs, or underscore characters. Conventions (and common sense) apply to this rule as well. When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases, it will also make your code self-documenting; fields named `cadence`, `speed`, and `gear`, for example, are much more intuitive than abbreviated versions, such as

s, c, and g. Also keep in mind that the name you choose must not be a keyword or reserved word.

- If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word. The names `gearRatio` and `currentGear` are prime examples of this convention. If your variable stores a constant value, such as `static final int NUM_GEARS = 6`, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

Primitive Data Types

The Java programming language is statically typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you've already seen:

```
int gear = 1;
```

Doing so tells your program that a field named *gear* exists, holds numerical data, and has an initial value of 1. A variable's data type determines the values it may contain plus the operations that may be performed on it. In addition to `int`, the Java programming language supports seven other *primitive data types*. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are as follows:

1. The **`byte`** data type is an 8-bit signed two's complement integer. It has a minimum value of –128 and a maximum value of 127 (inclusive). The `byte` data type can be useful for saving memory in large arrays, where the memory savings actually matters. It can also be used in place of `int` where its limits help clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
2. The **`short`** data type is a 16-bit signed two's complement integer. It has a minimum value of –32,768 and a maximum value of 32,767 (inclusive). As with `byte`, the same guidelines apply: you can use a `short` to save memory in large arrays in situations where the memory savings actually matter.
3. By default, the **`int`** data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31} - 1$. In Java SE 8 and later, you can use the `int` data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of $2^{32} - 1$. The `Integer` class also supports unsigned 32-bit integers. Static methods like

`compareUnsigned` and `divideUnsigned` have been added to the `Integer` class to support arithmetic operations for unsigned integers.[1]

4. The **`long`** data type is a 64-bit two's complement integer. The signed `long` has a minimum value of -2^{63} and a maximum value of $2^{63} - 1$. In Java SE 8 and later, you can use the `long` data type to represent an unsigned 64-bit `long`, which has a minimum value of 0 and a maximum value of $2^{64} - 1$. Use this data type when you need a range of values wider than those provided by the `int` data type. The `Long` class also contains methods like `compareUnsigned` and `divideUnsigned` to support arithmetic operations for unsigned `long` values.[2]
5. The **`float`** data type is a single-precision 32-bit IEEE 754 floating-point value. Its range of values is beyond the scope of this discussion but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification.[3] As with the recommendations for the `byte` and `short` data types, use a `float` (instead of `double`) value if you need to save memory in large arrays of floating-point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the `java.math.BigDecimal` class instead.[4] Chapter 9 covers `BigDecimal` and other useful classes provided by the Java platform.
6. The **`double`** data type is a double-precision 64-bit IEEE 754 floating-point value. Its range of values is beyond the scope of this discussion but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification.[5] For decimal values, this data type is generally the default choice. As mentioned previously, this data type should never be used for precise values, such as currency.
7. The **`boolean`** data type has only two possible values: `true` and `false`. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
8. The **`char`** data type is a single 16-bit Unicode character. It has a minimum value of `'\u0000'` (or 0) and a maximum value of `'\uffff'` (or 65,535 inclusive).

In addition to the eight primitive data types, the Java programming language also provides special support for character strings through the `java.lang.String`

1. `8/docs/api/java/lang/Integer.html`
2. `8/docs/api/java/lang/Long.html`
3. `specs/jls/se7/html/jls-4.html#jls-4.2.3`
4. `8/docs/api/java/math/BigDecimal.html`
5. `specs/jls/se7/html/jls-4.html#jls-4.2.3`

class.[6] Enclosing your character string within double quotes will automatically create a new `String` object—for example, `String s = "this is a string";`. `String` objects are *immutable*, which means that, once created, their values cannot be changed. The `String` class is not technically a primitive data type, but considering the special support given to it by the language, you'll probably tend to think of it as such. You'll learn more about the `String` class in Chapter 9.

Default Values

It's not always necessary to assign a value when a field is declared. Fields that are declared but not initialized will be set to a reasonable default by the compiler. Generally speaking, this default will be zero or `null`, depending on the data type. Relying on such default values, however, is generally considered bad programming style. Table 3.1 summarizes the default values for the above data types.

Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an uninitialized local variable will result in a compile-time error.

Literals

You may have noticed that the `new` keyword isn't used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class. A *literal* is the source code representation of a fixed value; literals are represented directly in your code without requiring computation. As shown here, it's possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

Integer Literals

An integer literal is of type `long` if it ends with the letter `L` or `l`; otherwise, it is of type `int`. It is recommended that you use the uppercase letter `L` because the lowercase letter `l` is hard to distinguish from the digit `1`.

Values of the integral types `byte`, `short`, `int`, and `long` can be created from `int` literals. Values of type `long` that exceed the range of `int` can be created from `long` literals. Integer literals can be expressed by these number systems:

6. `8/docs/api/java/lang/String.html`

Table 3.1 Default Values for Data Types

Data type	Default value (for fields)
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`
`boolean`	`false`
`char`	`'\u0000'`
`String` (or any object)	`null`

- *Decimal.* Base 10, whose digits consist of the numbers 0 through 9 (This is the number system you use every day.)
- *Hexadecimal.* Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F
- *Binary.* Base 2, whose digits consists of the numbers 0 and 1

For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need to use another number system, the following example shows the correct syntax. The prefix `0x` indicates hexadecimal and `0b` indicates binary:

```
// The number 26, in decimal
int decVal = 26;
// The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;
```

Floating-Point Literals

A floating-point literal is of type `float` if it ends with the letter `F` or `f`; otherwise, its type is `double` and it can optionally end with the letter `D` or `d`. The floating-point types (`float` and `double`) can also be expressed using `E` or `e` (for scientific notation), `F` or `f` (32-bit float literal), and `D` or `d` (64-bit double literal, which is the default and by convention is omitted).

```
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1 = 123.4f;
```

Character and String Literals

Literals of types `char` and `String` may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a *Unicode escape*, such as `'\u0108'` (for a capital C with circumflex, Ĉ) or `"S\u00ED Se\u00F1or"` (for *Sí Señor* in Spanish). Always use 'single quotes' for `char` literals and "double quotes" for `String` literals. Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in `char` or `String` literals.

The Java programming language also supports a few special escape sequences for `char` and `String` literals: `\b` (backspace), `\t` (tab), `\n` (line feed), `\f` (form feed), `\r` (carriage return), `\"` (double quote), `\'` (single quote), and `\\` (backslash).

There's also a special `null` literal that can be used as a value for any reference type. You may assign `null` to any variable except variables of primitive types. There's little you can do with a `null` value beyond testing for its presence. Therefore, `null` is often used in programs as a marker to indicate that some object is unavailable.

Finally, there's also a special kind of literal called a *class literal*, formed by taking a type name and appending `.class` (e.g., `String.class`). This refers to the object (of type `Class`) that represents the type itself.

Using Underscore Characters in Numeric Literals

Any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables you, for example, to separate groups of digits in numeric literals, which can improve the readability of your code.

For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma or a space as a separator.

The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

You can place underscores only between digits; you cannot place underscores in the following places:

- At the beginning or end of a number
- Adjacent to a decimal point in a floating-point literal
- Prior to an `F` or `L` suffix
- In positions where a string of digits is expected

The following examples demonstrate valid and invalid underscore placements (which are bold) in numeric literals:

```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;
// Invalid: cannot put underscores
// prior to an L suffix
long socialSecurityNumber1 = 999_99_9999_L;

// OK (decimal literal)
int x1 = 5_2;
// Invalid: cannot put underscores
// At the end of a literal
int x2 = 52_;
// OK (decimal literal)
int x3 = 5_______2;

// Invalid: cannot put underscores
// in the 0x radix prefix
int x4 = 0_x52;
// Invalid: cannot put underscores
// at the beginning of a number
int x5 = 0x_52;
// OK (hexadecimal literal)
int x6 = 0x5_2;
// Invalid: cannot put underscores
// at the end of a number
int x7 = 0x52_;
```

Arrays

An *array* is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. You have seen an example of arrays already, in the `main` method of the "Hello World!" application. This section discusses arrays in greater detail.

Each item in an array is called an *element*, and each element is accessed by its numerical *index*. As shown in the preceding illustration, numbering begins with 0. The ninth element, for example, would therefore be accessed at index 8.

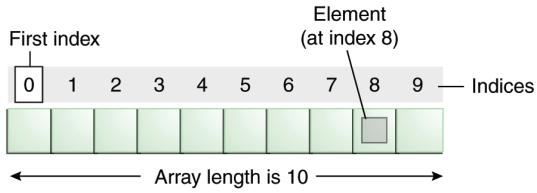


Figure 3.1 An Array of Ten Elements

The following program, `ArrayDemo`, creates an array of integers, puts some values in the array, and prints each value to standard output:

```
class ArrayDemo {
    public static void main(String[] args) {
        // declares an array of integers
        int[] anArray;

        // allocates memory for 10 integers
        anArray = new int[10];

        // initialize first element
        anArray[0] = 100;
        // initialize second element
        anArray[1] = 200;
        // and so forth
        anArray[2] = 300;
        anArray[3] = 400;
        anArray[4] = 500;
        anArray[5] = 600;
        anArray[6] = 700;
        anArray[7] = 800;
        anArray[8] = 900;
        anArray[9] = 1000;

        System.out.println("Element at index 0: "
                           + anArray[0]);
        System.out.println("Element at index 1: "
                           + anArray[1]);
        System.out.println("Element at index 2: "
                           + anArray[2]);
        System.out.println("Element at index 3: "
                           + anArray[3]);
        System.out.println("Element at index 4: "
                           + anArray[4]);
        System.out.println("Element at index 5: "
                           + anArray[5]);
        System.out.println("Element at index 6: "
                           + anArray[6]);
        System.out.println("Element at index 7: "
                           + anArray[7]);
        System.out.println("Element at index 8: "
                           + anArray[8]);
        System.out.println("Element at index 9: "
                           + anArray[9]);
    }
}
```

Here is the output from this program:

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000
```


In a real-world programming situation, you would probably use one of the supported *looping constructs* to iterate through each element of the array, rather than write each line individually as in the preceding example. However, the example clearly illustrates the array syntax. You will learn about the various looping constructs (`for`, `while`, and `do-while`) later in the "Control Flow" section.

Declaring a Variable to Refer to an Array

The preceding program declares an array (named `anArray`) with the following code:

```
// declares an array of integers
int[] anArray;
```

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written as *`type`*`[]`, where *`type`* is the data type of the contained elements; the brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty). An array's name can be anything you want, provided that it follows the rules and conventions as previously discussed in the "Naming" section. As with variables of other types, the declaration does not actually create an array; it simply tells the compiler that this variable will hold an array of the specified type. Similarly, you can declare arrays of other types:

```
byte[] anArrayOfBytes;
short[] anArrayOfShorts;
long[] anArrayOfLongs;
float[] anArrayOfFloats;
double[] anArrayOfDoubles;
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

You can also place the brackets after the array's name:

```
// this form is discouraged
float anArrayOfFloats[];
```

However, convention discourages this form; the brackets identify the array type and should appear with the type designation.

Creating, Initializing, and Accessing an Array

One way to create an array is with the `new` operator. The next statement in the `ArrayDemo` program allocates an array with enough memory for ten integer elements and assigns the array to the `anArray` variable:

```
// create an array of integers
anArray = new int[10];
```

If this statement is missing, then the compiler prints an error like the following and compilation fails:

```
ArrayDemo.java:4: Variable anArray may not have been initialized.
```

The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // and so forth
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = {
    100, 200, 300,
    400, 500, 600,
    700, 800, 900, 1000
};
```

Here the length of the array is determined by the number of values provided between braces and separated by commas.

You can also declare an array of arrays (also known as a *multidimensional* array) by using two or more sets of brackets, such as `String[][] names`. Each element, therefore, must be accessed by a corresponding number of index values.

In the Java programming language, a multidimensional array is an array whose components are themselves arrays. This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length, as shown in the following `MultiDimArrayDemo` program:

```
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String[][] names = {
            {"Mr. ", "Mrs. ", "Ms. "},
            {"Smith", "Jones"}
        };
        // Mr. Smith
        System.out.println(names[0][0] + names[1][0]);
        // Ms. Jones
        System.out.println(names[0][2] + names[1][1]);
    }
}
```


Here is the output from this program:

```
Mr. Smith
Ms. Jones
```

Finally, you can use the built-in `length` property to determine the size of any array. The following code prints the array's size to standard output:

```
System.out.println(anArray.length);
```

Copying Arrays

The `System` class has an `arraycopy()` method that you can use to efficiently copy data from one array into another:

```
public static void arraycopy(Object src, int srcPos,
                             Object dest, int destPos, int length)
```

The two `Object` arguments specify the array to copy *from* and the array to copy *to*. The three `int` arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

The following program, `ArrayCopyDemo`, declares an array of `char` elements, spelling the word *decaffeinated*. It uses the `System.arraycopy()` method to copy a subsequence of array components into a second array:

```
class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                            'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

Here is output from this program:

```
caffein
```

3

Array Manipulations

Arrays are a powerful and useful concept in programming. Java SE provides methods to perform some of the most common manipulations related to arrays. For instance, the `ArrayCopyDemo` example uses the `arraycopy()` method of the `System` class instead of manually iterating through the elements of the source array and placing each one into the destination array. This is performed behind the scenes, enabling the developer to use just one line of code to call the method.

For your convenience, Java SE provides several methods for performing array manipulations (common tasks such as copying, sorting, and searching arrays) in the `java.util.Arrays` class.[7] For instance, the previous example can be modified to use the `copyOfRange()` method of the `java.util.Arrays` class, as you can see in the `ArrayCopyOfDemo` example. The difference is that using the `copyOfRange()` method does not require you to create the destination array before calling the method because the destination array is returned by the method:

```
class ArrayCopyOfDemo {
    public static void main(String[] args) {

        char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e',
            'i', 'n', 'a', 't', 'e', 'd'};

        char[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);

        System.out.println(new String(copyTo));
    }
}
```

As you can see, the output from this program is the same (`caffein`), although it requires fewer lines of code.

Some other useful operations provided by methods in the `java.util.Arrays` class are as follows:

- Search an array for a specific value to get the index at which it is placed (the `binarySearch()` method).
- Compare two arrays to determine if they are equal or not (the `equals()` method).
- Fill an array to place a specific value at each index (the `fill()` method).

7. `8/docs/api/java/util/Arrays.html`

- Sort an array into ascending order. This can be done either sequentially, using the `sort()` method, or concurrently, using the `parallelSort()` method introduced in Java SE 8. Parallel sorting of large arrays on multiprocessor systems is faster than sequential array sorting.

Summary of Variables

The Java programming language uses both *fields* and *variables* as part of its terminology. Instance variables (nonstatic fields) are unique to each instance of a class. Class variables (static fields) are fields declared with the `static` modifier; there is exactly one copy of a class variable, regardless of how many times the class has been instantiated. Local variables store temporary state inside a method. Parameters are variables that provide extra information to a method; both local variables and parameters are always classified as *variables* (not *fields*). When naming your fields or variables, there are rules and conventions that you should (or must) follow.

The eight primitive data types are `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`. The `java.lang.String` class represents character strings.[8] The compiler will assign a reasonable default value for fields of these types; for local variables, a default value is never assigned. A literal is the source code representation of a fixed value. An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

Questions and Exercises: Variables

Questions

1. The term *instance variable* is another name for ___.
2. The term *class variable* is another name for ___.
3. A local variable stores temporary state; it is declared inside a ___.
4. A variable declared within the opening and closing parenthesis of a method signature is called a ___.
5. What are the eight primitive data types supported by the Java programming language?
6. Character strings are represented by the class ___.
7. An ___ is a container object that holds a fixed number of values of a single type.

8. `8/docs/api/java/lang/String.html`

Exercises

1. Create a small program that defines some fields. Try creating some illegal field names and see what kind of error the compiler produces. Use the naming rules and conventions as a guide.
2. In the program you created in Exercise 1, try leaving the fields uninitialized and print out their values. Try the same with a local variable and see what kind of compiler errors you can produce. Becoming familiar with common compiler errors will make it easier to recognize bugs in your code.

Answers

You can find answers to these questions and exercises at `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/answers_variables.html`.

Operators

Now that you've learned how to declare and initialize variables, you probably want to know how to *do something* with them. Learning the operators of the Java programming language is a good place to start. Operators are special symbols that perform specific operations on one, two, or three *operands* and then return a result.

As we explore the operators of the Java programming language, it may be helpful for you to know ahead of time which operators have the highest precedence. The operators in Table 3.2 are listed according to precedence order. The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower precedence. Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.

In general-purpose programming, certain operators tend to appear more frequently than others; for example, the assignment operator (=) is far more common than the unsigned right shift operator (>>>). With that in mind, the following discussion focuses first on the operators that you're most likely to use on a regular basis and ends focusing on those that are less common. Each discussion is accompanied by sample code that you can compile and run. Studying its output will help reinforce what you've just learned.

Table 3.2 Operator Precedence

Operators	Precedence
Postfix	*expr* ++ *expr* --
unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Assignment, Arithmetic, and Unary Operators

The Simple Assignment Operator

One of the most common operators that you'll encounter is the simple assignment operator, =. You saw this operator in the `Bicycle` class; it assigns the value on its right to the operand on its left:

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

This operator can also be used on objects to assign *object references*, as discussed in Chapter 4, "Creating Objects."

The Arithmetic Operators

The Java programming language provides operators that perform addition, subtraction, multiplication, and division. There's a good chance you'll recognize them by their counterparts in basic mathematics. The only symbol that might look new to you is `%`, which divides one operand by another and returns the remainder as its result.

Table 3.3 Arithmetic Operators

Operator	Description
+	Additive operator (also used for String concatenation)
–	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator

The following program, `ArithmeticDemo`, tests the arithmetic operators:

```
class ArithmeticDemo {

    public static void main (String[] args) {
        int result = 1 + 2;
        // result is now 3
        System.out.println("1 + 2 = " + result);
        int original_result = result;

        result = result - 1;
        // result is now 2
        System.out.println(original_result + " - 1 = " + result);
        original_result = result;

        result = result * 2;
        // result is now 4
        System.out.println(original_result + " * 2 = " + result);
        original_result = result;

        result = result / 2;
        // result is now 2
        System.out.println(original_result + " / 2 = " + result);
        original_result = result;

        result = result + 8;
        // result is now 10
        System.out.println(original_result + " + 8 = " + result);
        original_result = result;

        result = result % 7;
        // result is now 3
        System.out.println(original_result + " % 7 = " + result);
    }
}
```

This program prints the following:

```
1 + 2 = 3
3 - 1 = 2
2 * 2 = 4
4 / 2 = 2
2 + 8 = 10
10 % 7 = 3
```

You can also combine the arithmetic operators with the simple assignment operator to create *compound assignments*. For example, `x+=1;` and `x=x+1;` both increment the value of `x` by 1.

The `+` operator can also be used for concatenating (joining) two strings together, as shown in the following `ConcatDemo` program:


```
class ConcatDemo {
    public static void main(String[] args){
        String firstString = "This is";
        String secondString = " a concatenated string.";
        String thirdString = firstString + secondString;
        System.out.println(thirdString);
    }
}
```

By the end of this program, the variable `thirdString` contains `"This is a concatenated string."`, which gets printed to standard output.

The Unary Operators

The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a `boolean`.

The following program, `UnaryDemo`, tests the unary operators:

```
class UnaryDemo {

    public static void main(String[] args) {

        int result = +1;
        // result is now 1
        System.out.println(result);

        result--;
        // result is now 0
        System.out.println(result);

        result++;
        // result is now 1
        System.out.println(result);

        result = -result;
        // result is now -1
        System.out.println(result);

        boolean success = false;
        // false
        System.out.println(success);
        // true
        System.out.println(!success);
    }
}
```

Table 3.4 Unary Operators

Operator	Description
+	Unary plus operator; indicates positive value (numbers are positive without this, however)
–	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a `boolean`

The increment/decrement operators can be applied before (prefix) or after (postfix) the operand. The code `result++;` and `++result;` will both end in `result` being incremented by one. The only difference is that the prefix version (`++result`) evaluates to the incremented value, whereas the postfix version (`result++`) evaluates to the original value. If you are just performing a simple increment/decrement operation, it doesn't really matter which version you choose. But if you use this operator in part of a larger expression, the one you choose may make a significant difference.

The following program, `PrePostDemo`, illustrates the prefix/postfix unary increment operator:

```
class PrePostDemo {
    public static void main(String[] args){
        int i = 3;
        i++;
        // prints 4
        System.out.println(i);
        ++i;
        // prints 5
        System.out.println(i);
        // prints 6
        System.out.println(++i);
        // prints 6
        System.out.println(i++);
        // prints 7
        System.out.println(i);
    }
}
```

Equality, Relational, and Conditional Operators

The Equality and Relational Operators

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. The majority of these operators will probably look familiar to you as well. Keep in mind that you must use ==, not =, when testing if two primitive values are equal:

```
== equal to
!= not equal to
> greater than
>= greater than or equal to
< less than
<= less than or equal to
```


The following program, `ComparisonDemo`, tests the comparison operators:

```
class ComparisonDemo {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if(value1 == value2)
            System.out.println("value1 == value2");
        if(value1 != value2)
            System.out.println("value1 != value2");
        if(value1 > value2)
            System.out.println("value1 > value2");
        if(value1 < value2)
            System.out.println("value1 < value2");
        if(value1 <= value2)
            System.out.println("value1 <= value2");
    }
}
```

Here is the output:

```
value1 != value2
value1 < value2
value1 <= value2
```

The Conditional Operators

The `&&` and `||` operators perform *Conditional-AND* and *Conditional-OR* operations on two `boolean` expressions. These operators exhibit *short-circuiting* behavior, which means that the second operand is evaluated only if needed:

```
&& Conditional-AND
|| Conditional-OR
```

The following program, `ConditionalDemo1`, tests these operators:

```
class ConditionalDemo1 {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if((value1 == 1) && (value2 == 2))
            System.out.println("value1 is 1 AND value2 is 2");
        if((value1 == 1) || (value2 == 1))
```

```
            System.out.println("value1 is 1 OR value2 is 1");
    }
}
```


Another conditional operator is `?:`, which can be thought of as shorthand for an `if-then-else` statement (discussed in the "Control Flow Statements" section of this chapter). This operator is also known as the *ternary operator* because it uses three operands. In the following example, this operator should be read as follows: "If `someCondition` is `true`, assign the value of `value1` to `result`. Otherwise, assign the value of `value2` to `result`."

The following program, `ConditionalDemo2`, tests the `?:` operator:

```
class ConditionalDemo2 {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        int result;
        boolean someCondition = true;
        result = someCondition ? value1 : value2;

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

Because `someCondition` is true, this program prints `1` to the screen. Use the `?:` operator instead of an `if-then-else` statement if it makes your code more readable (e.g., when the expressions are compact and without side effects, such as in assignments).

The Type Comparison Operator instanceof

The `instanceof` operator compares an object to a specified type. You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

The following program, `InstanceofDemo`, defines a parent class (named `Parent`), a simple interface (named `MyInterface`), and a child class (named `Child`) that inherits from the parent and implements the interface.

```
class InstanceofDemo {
    public static void main(String[] args) {

        Parent obj1 = new Parent();
        Parent obj2 = new Child();

        System.out.println("obj1 instanceof Parent: "
            + (obj1 instanceof Parent));
        System.out.println("obj1 instanceof Child: "
            + (obj1 instanceof Child));
```

```
        System.out.println("obj1 instanceof MyInterface: "
            + (obj1 instanceof MyInterface));
        System.out.println("obj2 instanceof Parent: "
            + (obj2 instanceof Parent));
        System.out.println("obj2 instanceof Child: "
            + (obj2 instanceof Child));
        System.out.println("obj2 instanceof MyInterface: "
            + (obj2 instanceof MyInterface));
    }
}

class Parent {}
class Child extends Parent implements MyInterface {}
interface MyInterface {}
```

Here is the output:

```
obj1 instanceof Parent: true
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
obj2 instanceof Child: true
obj2 instanceof MyInterface: true
```

When using the `instanceof` operator, keep in mind that `null` is not an instance of anything.

Bitwise and Bit Shift Operators

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. The operators discussed in this section are less commonly used. Therefore their coverage is brief; the intent is to simply make you aware that these operators exist.

The unary bitwise complement operator (~) inverts a bit pattern; it can be applied to any of the integral types, making every 0 a 1 and every 1 a 0. For example, a `byte` contains 8 bits; applying this operator to a value whose bit pattern is 00000000 would change its pattern to 11111111.

The signed left shift operator (<<) shifts a bit pattern to the left, and the signed right shift operator (>>) shifts a bit pattern to the right. The bit pattern is given by the left-hand operand, and the number of positions to shift is given by the right-hand operand. The unsigned right shift operator (>>>) shifts a zero into the leftmost position, while the leftmost position after >> depends on sign extension.

The bitwise `&` operator performs a bitwise AND operation. The bitwise ^ operator performs a bitwise exclusive OR operation. The bitwise | operator performs a bitwise inclusive OR operation.

The following program, `BitDemo`, uses the bitwise AND operator to print the number 2 to standard output:

```
class BitDemo {
    public static void main(String[] args) {
        int bitmask = 0x000F;
        int val = 0x2222;
        // prints "2"
        System.out.println(val & bitmask);
    }
}
```


Summary of Operators

The following quick reference summarizes the operators supported by the Java programming language.

Simple Assignment Operator

= Simple assignment operator

Arithmetic Operators

+ Additive operator (also used for `String` concatenation)
- Subtraction operator
* Multiplication operator
/ Division operator
% Remainder operator

Unary Operators

+ Unary plus operator; indicates positive value, although numbers can be positive without this
- Unary minus operator; negates an expression
++ Increment operator; increments a value by 1
-- Decrement operator; decrements a value by 1
! Logical complement operator; inverts the value of a `boolean`

Equality and Relational Operators

== Equal to
!= Not equal to
> Greater than
>= Greater than or equal to
< Less than
<= Less than or equal to

Conditional Operators

`&&` Conditional AND
`||` Conditional OR
`?:` Ternary (shorthand for `if-then-else` statement)

Type Comparison Operator

`instanceof` Compares an object to a specified type

Bitwise and Bit Shift Operators

`~` Unary bitwise complement
`<<` Signed left shift
`>>` Signed right shift
`>>>` Unsigned right shift
`&` Bitwise AND
`^` Bitwise exclusive OR
`|` Bitwise inclusive OR

Questions and Exercises: Operators

Questions

1. Consider the following code snippet:

   ```
   arrayOfInts[j] > arrayOfInts[j+1]
   ```

   Which operators does the code contain?
2. Consider the following code snippet.

   ```
   int i = 10;
   int n = i++%5;
   ```

   a. What are the values of `i` and `n` after the code is executed?
   b. What are the final values of `i` and `n` if instead of using the postfix increment operator (`i++`), you use the prefix version (`++i`)?
3. To invert the value of a `boolean`, which operator would you use?
4. Which operator is used to compare two values, `=` or `==` ?
5. Explain the following code sample: `result = someCondition ? value1 : value2;`

Exercises

1. Change the following program to use compound assignments:

```
class ArithmeticDemo {

     public static void main (String[] args){

          int result = 1 + 2; // result is now 3
          System.out.println(result);

          result = result - 1; // result is now 2
          System.out.println(result);

          result = result * 2; // result is now 4
          System.out.println(result);

          result = result / 2; // result is now 2
          System.out.println(result);

          result = result + 8; // result is now 10
          result = result % 7; // result is now 3
          System.out.println(result);
     }
}
```

2. In the following program, explain why the value 6 is printed twice in a row:

```
class PrePostDemo {
    public static void main(String[] args){
        int i = 3;
        i++;
        System.out.println(i); // "4"
        ++i;
        System.out.println(i); // "5"
        System.out.println(++i); // "6"
        System.out.println(i++); // "6"
        System.out.println(i); // "7"
    }
}
```

Answers

You can find answers to these questions and exercises at `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/answers_operators.html`.

Expressions, Statements, and Blocks

Now that you understand variables and operators, it's time to learn about *expressions*, *statements*, and *blocks*. Operators may be used in building expressions, which compute values. Expressions are the core components of statements, and statements may be grouped into blocks.

Expressions

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language that evaluates

to a single value. You've already seen examples of expressions, illustrated in bold as follows:

```
int cadence = 0;
anArray[0] = 100;
System.out.println("Element 1 at index 0: " + anArray[0]);

int result = 1 + 2; // result is now 3
if (value1 == value2)
    System.out.println("value1 == value2");
```


The data type of the value returned by an expression depends on the elements used in the expression. The expression `cadence = 0` returns an `int` because the assignment operator returns a value of the same data type as its left-hand operand; in this case, `cadence` is an `int`. As you can see from the other expressions, an expression can return other types of values as well, such as `boolean` or `String`.

The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other. Here's an example of a compound expression:

```
1 * 2 * 3
```

In this particular example, the order in which the expression is evaluated is unimportant because the result of multiplication is independent of order; the outcome is always the same, regardless of the order of the numbers being multiplied. However, this is not true for all expressions. For example, the following expression gives different results, depending on whether you perform the addition or the division operation first:

```
x + y / 100 // ambiguous
```

You can specify exactly how an expression will be evaluated using balanced parentheses: `(` and `)`. For example, to make the previous expression unambiguous, you could write the following:

```
(x + y) / 100 // unambiguous, recommended
```

If you don't explicitly indicate the order for the operations to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators that have a higher precedence get evaluated first. For example, the division operator has a higher precedence than the addition operator. Therefore the following two statements are equivalent:

```
x + y / 100

x + (y / 100) // unambiguous, recommended
```


When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first. This practice makes code easier to read and maintain.

Statements

Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (`;`):

- Assignment expressions
- Any use of `++` or `--`
- Method invocations
- Object creation expressions

Such statements are called *expression statements*. Here are some examples of expression statements:

```
// assignment statement
aValue = 8933.234;
// increment statement
aValue++;
// method invocation statement
System.out.println("Hello World!");
// object creation statement
Bicycle myBike = new Bicycle();
```

In addition to expression statements, there are two other kinds of statements: *declaration statements* and *control flow statements*. A *declaration statement* declares a variable. You've seen many examples of declaration statements already:

```
// declaration statement
double aValue = 8933.234;
```

Finally, *control flow statements* regulate the order in which statements get executed. You'll learn about control flow statements in the next section, "Control Flow Statements."

Blocks

A *block* is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following example, `BlockDemo`, illustrates the use of blocks:


```
class BlockDemo {
     public static void main(String[] args) {
          boolean condition = true;
          if (condition) { // begin block 1
               System.out.println("Condition is true.");
          } // end block one
          else { // begin block 2
               System.out.println("Condition is false.");
          } // end block 2
     }
}
```

Questions and Exercises: Expressions, Statements, and Blocks

Questions

1. Operators may be used in building ___, which compute values.
2. Expressions are the core components of ___.
3. Statements may be grouped into ___.
4. The following code snippet is an example of a ___ expression:

   ```
   1 * 2 * 3
   ```

5. Statements are roughly equivalent to sentences in natural languages, but instead of ending with a period, a statement ends with a ___.
6. A block is a group of zero or more statements between balanced ___ and can be used anywhere a single statement is allowed.

Exercise

1. Identify the following kinds of expression statements:
   - `aValue = 8933.234;`
   - `aValue++;`
   - `System.out.println("Hello World!");`
   - `Bicycle myBike = new Bicycle();`

Answers

You can find answers to these questions and exercises at `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/answers_expressions.html`.

Control Flow Statements

The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code. This section describes the decision-making statements (`if-then`, `if-then-else`, `switch`), the looping statements (`for`, `while`, `do-while`), and the branching statements (`break`, `continue`, `return`) supported by the Java programming language.

The if-then and if-then-else Statements

The if-then Statement

The `if-then` statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to `true`. For example, the `Bicycle` class could allow the brakes to decrease the bicycle's speed *only if* the bicycle is already in motion. One possible implementation of the `applyBrakes` method could be as follows:

```
void applyBrakes() {
    // the "if" clause: bicycle must be moving
    if (isMoving){
        // the "then" clause: decrease current speed
        currentSpeed--;
    }
}
```

If this test evaluates to `false` (meaning that the bicycle is not in motion), control jumps to the end of the `if-then` statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

```
void applyBrakes() {
    // same as above, but without braces
    if (isMoving)
        currentSpeed--;
}
```

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause,

a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

The if-then-else Statement

The `if-then-else` statement provides a secondary path of execution when an "if" clause evaluates to `false`. You could use an `if-then-else` statement in the `applyBrakes` method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
void applyBrakes() {
    if (isMoving) {
        currentSpeed--;
    } else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

The following program, `IfElseDemo`, assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on:

```
class IfElseDemo {
    public static void main(String[] args) {

        int testscore = 76;
        char grade;

        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
    }
}
```

The output from the program is as follows:

```
Grade = C
```

You may have noticed that the value of `testscore` can satisfy more than one expression in the compound statement: `76 >= 70` and `76 >= 60`. However, once a condition is satisfied, the appropriate statements are executed (`grade = 'C';`) and the remaining conditions are not evaluated.

The switch Statement

Unlike `if-then` and `if-then-else` statements, the `switch` statement can have a number of possible execution paths. A `switch` works with the `byte`, `short`, `char`, and `int` primitive data types. It also works with *enumerated types* (discussed in Chapter 4, "Enum Types"), the `String` class, and a few special classes that wrap certain primitive types: `Character`, `Byte`, `Short`, and `Integer` (discussed in Chapter 9).

The following code example, `SwitchDemo`, declares an `int` named `month` whose value represents a month. The code displays the name of the month, based on the value of `month`, using the `switch` statement:

```
public class SwitchDemo {
    public static void main(String[] args) {

        int month = 8;
        String monthString;
        switch (month) {
            case 1: monthString = "January";
                    break;
            case 2: monthString = "February";
                    break;
            case 3: monthString = "March";
                    break;
            case 4: monthString = "April";
                    break;
            case 5: monthString = "May";
                    break;
            case 6: monthString = "June";
                    break;
            case 7: monthString = "July";
                    break;
            case 8: monthString = "August";
                    break;
            case 9: monthString = "September";
                    break;
            case 10: monthString = "October";
                    break;
            case 11: monthString = "November";
                    break;
            case 12: monthString = "December";
                    break;
            default: monthString = "Invalid month";
                    break;
        }
        System.out.println(monthString);
    }
}
```

In this case, `August` is printed to standard output.

The body of a `switch` statement is known as a *switch block*. A statement in the `switch` block can be labeled with one or more `case` or `default` labels. The `switch`

statement evaluates its expression and then executes all statements that follow the matching `case` label.

You could also display the name of the month with `if-then-else` statements:


```
int month = 8;
if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
}
// ... and so on
```

The choice between `if-then-else` statements or a `switch` statement depends on readability and the expression that the statement is testing. An `if-then-else` statement can test expressions based on ranges of values or conditions, whereas a `switch` statement tests expressions based only on a single integer, enumerated value, or `String` object.

Another point of interest is the `break` statement. Each `break` statement terminates the enclosing `switch` statement. Control flow continues with the first statement following the `switch` block. The `break` statements are necessary because without them, statements in `switch` blocks *fall through*: All statements after the matching `case` label are executed in sequence, regardless of the expression of subsequent `case` labels, until a `break` statement is encountered. The program `SwitchDemoFallThrough` shows statements in a `switch` block that fall through; it displays the month corresponding to the integer `month` and the months that follow in the year:

```
public class SwitchDemoFallThrough {
    public static void main(String[] args) {
        java.util.ArrayList<String> futureMonths =
            new java.util.ArrayList<String>();

        int month = 8;

        switch (month) {
            case 1: futureMonths.add("January");
            case 2: futureMonths.add("February");
            case 3: futureMonths.add("March");
            case 4: futureMonths.add("April");
            case 5: futureMonths.add("May");
            case 6: futureMonths.add("June");
            case 7: futureMonths.add("July");
            case 8: futureMonths.add("August");
            case 9: futureMonths.add("September");
            case 10: futureMonths.add("October");
            case 11: futureMonths.add("November");
            case 12: futureMonths.add("December");
                     break;
            default: break;
        }
```

```
            if (futureMonths.isEmpty()) {
                System.out.println("Invalid month number");
            } else {
                for (String monthName : futureMonths) {
                   System.out.println(monthName);
                }
            }
        }
    }
```

This is the output from the code:

```
August
September
October
November
December
```

Technically, the final `break` is not required because flow falls out of the `switch` statement. Using a `break` is recommended so that modifying the code is easier and less error prone. The `default` section handles all values that are not explicitly handled by one of the `case` sections.

The following code example, `SwitchDemo2`, shows how a statement can have multiple `case` labels. The code example calculates the number of days in a particular month:

```
class SwitchDemo2 {
    public static void main(String[] args) {

        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case 1: case 3: case 5:
            case 7: case 8: case 10:
            case 12:
                numDays = 31;
                break;
            case 4: case 6:
            case 9: case 11:
                numDays = 30;
                break;
            case 2:
                if (((year % 4 == 0) &&
                     !(year % 100 == 0))
                     || (year % 400 == 0))
                    numDays = 29;
                else
                    numDays = 28;
                break;
            default:
                System.out.println("Invalid month.");
```

```
                break;
            }
            System.out.println("Number of Days = "
                               + numDays);
        }
    }
```


This is the output from the code:

```
Number of Days = 29
```

Using Strings in switch Statements

You can use a `String` object in the `switch` statement's expression. The following code example, `StringSwitchDemo`, displays the number of the month based on the value of the `String` named `month`:

```
public class StringSwitchDemo {

    public static int getMonthNumber(String month) {

        int monthNumber = 0;

        if (month == null) {
            return monthNumber;
        }

        switch (month.toLowerCase()) {
            case "january":
                monthNumber = 1;
                break;
            case "february":
                monthNumber = 2;
                break;
            case "march":
                monthNumber = 3;
                break;
            case "april":
                monthNumber = 4;
                break;
            case "may":
                monthNumber = 5;
                break;
            case "june":
                monthNumber = 6;
                break;
            case "july":
                monthNumber = 7;
                break;
            case "august":
                monthNumber = 8;
                break;
            case "september":
                monthNumber = 9;
                break;
```

```
            case "october":
                monthNumber = 10;
                break;
            case "november":
                monthNumber = 11;
                break;
            case "december":
                monthNumber = 12;
                break;
            default:
                monthNumber = 0;
                break;
        }

        return monthNumber;
    }

    public static void main(String[] args) {

        String month = "August";

        int returnedMonthNumber =
            StringSwitchDemo.getMonthNumber(month);

        if (returnedMonthNumber == 0) {
            System.out.println("Invalid month");
        } else {
            System.out.println(returnedMonthNumber);
        }
    }
}
```

The output from this code is `8`.

The `String` in the `switch` expression is compared with the expressions associated with each `case` label, as if the `String.equals`[9] method was being used. In order for the `StringSwitchDemo` example to accept any month regardless of case, `month` is converted to lowercase (with the `toLowerCase`[10] method) and all the strings associated with the `case` labels are in lowercase.

Note

This example checks if the expression in the `switch` statement is `null`. Ensure that the expression in any `switch` statement is not `null` to prevent a `NullPointerException` from being thrown.

9. `8/docs/api/java/lang/String.html#equals-java.lang.Object-`

10. `8/docs/api/java/lang/String.html#toLowerCase--`

The while and do-while Statements

The `while` statement continually executes a block of statements while a particular condition is `true`. Its syntax can be expressed as follows:

```
while (expression) {
     statement(s)
}
```

The `while` statement evaluates *`expression`*, which must return a `boolean` value. If the expression evaluates to `true`, the `while` statement executes the *`statement(s)`* in the `while` block. The `while` statement continues testing the expression and executing its block until the expression evaluates to `false`. Using the `while` statement to print the values from 1 through 10 can be accomplished via the following `WhileDemo` program:

```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}
```

You can implement an infinite loop using the `while` statement as follows:

```
while (true){
    // your code goes here
}
```

The Java programming language also provides a `do-while` statement, which can be expressed as follows:

```
do {
     statement(s)
} while (expression);
```

The difference between `do-while` and `while` is that `do-while` evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the `do` block are always executed at least once, as shown in the following `DoWhileDemo` program:

```
class DoWhileDemo {
    public static void main(String[] args){
        int count = 1;
        do {
```

```
            System.out.println("Count is: " + count);
            count++;
        } while (count < 11);
    }
}
```


The for Statement

The `for` statement provides a compact way to iterate over a range of values. Programmers often refer to it as the *for loop* because of the way it repeatedly loops until a particular condition is satisfied. The general form of the `for` statement can be expressed as follows:

```
for (initialization; termination; increment) {
    statement(s)
}
```

When using this version of the `for` statement, keep the following in mind:

- The *initialization* expression initializes the loop; it's executed once as the loop begins.
- When the *termination* expression evaluates to `false`, the loop terminates.
- The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

The following program, `ForDemo`, uses the general form of the `for` statement to print the numbers 1 through 10 to standard output:

```
class ForDemo {
    public static void main(String[] args){
         for(int i=1; i<11; i++){
              System.out.println("Count is: " + i);
         }
    }
}
```

Here is the output of this program:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Notice how the code declares a variable within the initialization expression. The scope of this variable extends from its declaration to the end of the block governed by the `for` statement, so it can be used in the termination and increment expressions as well. If the variable that controls a `for` statement is not needed outside the loop, it's best to declare the variable in the initialization expression. The names `i`, `j`, and `k` are often used to control `for` loops; declaring them within the initialization expression limits their life span and reduces errors.

The three expressions of the `for` loop are optional; an infinite loop can be created as follows:

```
// infinite loop
for ( ; ; ) {

    // your code goes here
}
```

The `for` statement also has another form designed for iteration through collections and arrays. This form is sometimes referred to as the *enhanced for* statement and can be used to make your loops more compact and easier to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
```

The following program, `EnhancedForDemo`, uses the enhanced `for` to loop through the array:

```
class EnhancedForDemo {
    public static void main(String[] args){
         int[] numbers =
             {1,2,3,4,5,6,7,8,9,10};
         for (int item : numbers) {
             System.out.println("Count is: " + item);
         }
    }
}
```

In this example, the variable `item` holds the current value from the numbers array. The output from this program is the same as before:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

We recommend using this form of the `for` statement instead of the general form whenever possible.

Branching Statements

The break Statement

The `break` statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the `switch` statement. You can also use an unlabeled `break` to terminate a `for`, `while`, or `do-while` loop, as shown in the following `BreakDemo` program:

```
class BreakDemo {
    public static void main(String[] args) {

        int[] arrayOfInts =
            { 32, 87, 3, 589,
              12, 1076, 2000,
              8, 622, 127 };
        int searchfor = 12;

        int i;
        boolean foundIt = false;

        for (i = 0; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at index " + i);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

This program searches for the number 12 in an array. The `break` statement, shown in boldface, terminates the `for` loop when that value is found. Control flow then transfers to the statement after the `for` loop. This program's output is as follows:

```
Found 12 at index 4
```

An unlabeled `break` statement terminates the innermost `switch`, `for`, `while`, or `do-while` statement, but a labeled `break` terminates an outer statement. The following program, `BreakWithLabelDemo`, is similar to the previous program but uses nested `for` loops to search for a value in a two-dimensional array. When the value is found, a labeled `break` terminates the outer `for` loop (labeled `search`):

```
class BreakWithLabelDemo {
    public static void main(String[] args) {

        int[][] arrayOfInts = {
            { 32, 87, 3, 589 },
            { 12, 1076, 2000, 8 },
            { 622, 127, 77, 955 }
        };
        int searchfor = 12;

        int i;
        int j = 0;
        boolean foundIt = false;

    search:
        for (i = 0; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length;
                 j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at " + i + ", " + j);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

This is the output of the program:

```
Found 12 at 1, 0
```

The `break` statement terminates the labeled statement; it does not transfer the flow of control to the label. Control flow is transferred to the statement immediately following the labeled (terminated) statement.

The continue Statement

The `continue` statement skips the current iteration of a `for`, `while`, or `do-while` loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the `boolean` expression that controls the loop. The following program, `ContinueDemo`, steps through a `String`, counting the occurrences of the letter *p*. If the current character is not a *p*, the `continue` statement skips the rest of the loop and proceeds to the next character. If it is a *p*, the program increments the letter count:

```
class ContinueDemo {
    public static void main(String[] args) {
```

```
        String searchMe = "peter piper picked a " + "peck of pickled peppers";
        int max = searchMe.length();
        int numPs = 0;

        for (int i = 0; i < max; i++) {
            // interested only in p's
            if (searchMe.charAt(i) != 'p')
                continue;

            // process p's
            numPs++;
        }
        System.out.println("Found " + numPs + " p's in the string.");
    }
}
```

Here is the output of this program:

```
Found 9 p's in the string.
```

To see this effect more clearly, try removing the `continue` statement and recompiling. When you run the program again, the count will be wrong, saying that it found 35 *p*'s instead of 9.

A labeled `continue` statement skips the current iteration of an outer loop marked with the given label. The following example program, `ContinueWithLabelDemo`, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and another to iterate over the string being searched. The following program, `ContinueWithLabelDemo`, uses the labeled form of the `continue` statement to skip an iteration in the outer loop:

```
class ContinueWithLabelDemo {
    public static void main(String[] args) {

        String searchMe = "Look for a substring in me";
        String substring = "sub";
        boolean foundIt = false;

        int max = searchMe.length() -
                  substring.length();

    test:
        for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (searchMe.charAt(j++) != substring.charAt(k++)) {
                    continue test;
                }
            }
```

```
                foundIt = true;
                    break test;
            }
            System.out.println(foundIt ? "Found it" : "Didn't find it");
        }
    }
```


Here is the output from this program:

```
Found it
```

The return Statement

The last of the branching statements is the `return` statement. The `return` statement exits from the current method, and control flow returns to where the method was invoked. The `return` statement has two forms: one that returns a value and another that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the `return` keyword:

```
return ++count;
```

The data type of the returned value must match the type of the method's declared return value. When a method is declared `void`, use the form of `return` that doesn't return a value:

```
return;
```

Chapter 4 covers everything you need to know about writing methods.

Summary of Control Flow Statements

The `if-then` statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to `true`. The `if-then-else` statement provides a secondary path of execution when an "if" clause evaluates to `false`. Unlike `if-then` and `if-then-else`, the `switch` statement allows for any number of possible execution paths. The `while` and `do-while` statements continually execute a block of statements while a particular condition is `true`. The difference between `do-while` and `while` is that `do-while` evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the `do` block are always executed at least once. The `for` statement provides a compact way to iterate over a range of values. It has two forms, one of which was designed for looping through collections and arrays.

Questions and Exercises: Control Flow Statements

Questions

1. The most basic control flow statement supported by the Java programming language is the ___ statement.
2. The ___ statement allows for any number of possible execution paths.
3. The ___ statement is similar to the `while` statement but evaluates its expression at the ___ of the loop.
4. How do you write an infinite loop using the `for` statement?
5. How do you write an infinite loop using the `while` statement?

Exercises

1. Consider the following code snippet:

```
if (aNumber >= 0)
    if (aNumber == 0)
        System.out.println("first string");
else System.out.println("second string");
System.out.println("third string");
```

   a. What output do you think the code will produce if `aNumber` is 3?
   b. Write a test program containing the previous code snippet; make `aNumber` 3. What is the output of the program? Is it what you predicted? Explain why the output is what it is; in other words, what is the control flow for the code snippet?
   c. Using only spaces and line breaks, reformat the code snippet to make the control flow easier to understand.
   d. Use braces, `{` and `}`, to further clarify the code.

Answers

You can find answers to these questions and exercises at `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/answers_flow.html`.

This page intentionally left blank

Index

E

G

M

This page intentionally left blank