An Introduction to Computer Science

Robert Sedgewick
Kevin Wayne

Princeton University

Preface

Computer science as an academic discipline has evolved to embrace a set of intellectual challenges on a par with other sciences. This fact, combined with the undeniable impact of computer science on the modern world, demands an introductory college textbook comparable with commonly-used textbooks in physics, chemistry, or biology. Accordingly, this book is intended to meet the need for an introductory college text in computer science. The distinctive feature of the book is that it has broader coverage of the field than is found in many texts that are currently in use.

All college students can benefit from exposure to computer science early in the curriculum. Therefore, increasing numbers of colleges and universities are requiring that students (particularly those in the sciences, engineering, mathematics, and even the social sciences) take one or two semester courses in computer science in the first year. In more mature disciplines, it is commonly accepted that, at this critical point in the curriculum, it is important to challenge students with fundamental intellectual issues while surveying the field. This book aims to support an introductory course that does so for computer science.

We have three primary goals. First, we want to *demystify* computer systems by unpeeling levels of abstraction down to the simplest physical component, leaving no black boxes. Second, we want to *empower* students by giving them the experience and insight necessary to exploit available technology whenever appropriate. Third, we want to build *awareness* of the substantial intellectual underpinnings of the field and its broad reach into other sciences.

The book is a self-contained treatment intended for people with no previous experience in computer science. While its primary purpose is to serve as a textbook for first-year college students, it may also serve as a broad introduction to the field that may be of interest to anyone who has not been exposed to its fundamentals.

Coverage

The book is organized around four areas of computer science: programming, architecture, theory, and systems. We introduce fundamental concepts in each area and pay special attention to relationships among them. A proper introduction to the field must do justice to each of these four areas. We also cover applications in scientific computing throughout the book, to reinforce and supplement the curriculum that is typically taught to students in mathematics, science, and engineering in high school and their first year in college.

Programming is essential to being able to understand and appreciate computer science. We cover basic programming in Java, abstract data types and Java classes, elementary data structures, and the design and analysis of algorithms. To teach these concepts, we use sample programs that solve interesting problems supported with exercises ranging in difficulty from self-study drills to challenging problems that call for creative solutions. Whenever possible, we draw examples from concepts to be covered elsewhere in the book or from scientific or commercial applications. Students who learn the programs in this book will have the solid foundation necessary to prepare them to be able to effectively use computers as they pursue any academic discipline.

Architecture refers both to the art of designing computers and to the process of building them. Our coverage of this topic is centered around an imaginary machine that is similar to real computers. We specify the machine in full detail, consider machine-language programs for familiar tasks, and present a Java simulator for the machine. We continue with a treatment of circuits and logical design, culminating in a description of how such a machine might be built from the ground up. Our intent is to demystify this apparently formidable challenge while also providing a context for under-

standing how Java programming and other familiar high-level tasks relate to actual machines.

Theory helps us to understand the limits on what we can accomplish with computers. We present Turing's classical results that show how simple abstract machines can help us to pose fundamental questions about the nature of computation. Some of these are among the most important scientific questions of our time. We also consider practical applications such as how to estimate the running times of our programs and how to design efficient algorithms.

Systems enable us to work with computers at a high level of abstraction. We describe the basic components of computer systems that support programming; operating systems for interacting with our programs and our data; networks that allow interaction among computers; and applications systems that provide specialized support for particular tasks.

Our coverage of the four areas is intertwined and also threaded with descriptions and examples of various classical algorithms, programming languages, scientific computing, and commercial applications. Where relevant, we also provide proper historical perspective.

Generally speaking, we *introduce* material that is covered, at most colleges and universities, in several later computer science courses. For computer science majors, this breadth of coverage provides a foundation to be reinforced in later courses. For students in other fields who have a chance to take only one or two courses in computer science, this introduction to programming, architecture, theory and systems in the context of scientific applications gives the background that they need to effectively address the impact of computer science on their chosen fields.

Use in the Curriculum

This book is intended for a college course aimed at teaching novices to program while at the same time introducing them to the field of computer science. As such, its content fulfills the requirements of a first course in computer science in a reasonable way. But it does offer an alternative to many courses that are traditionally taught. Rather than delving deeply into the details of a particular programming language, we put programming in con-

text. Rather than giving scant coverage to programming while surveying computer science, we are able to address fundamental issues that cannot be understood without programming experience.

One option is to use this book is as the basis for a full-year college course, perhaps supplemented with a standard text on computer programming. Depending on the needs of the students and the experience of the instructor, diversions of varying length on basic programming skills are appropriate. For logical consistency, we have put most of the material on programming at the beginning of the book; in practice, it is best to spread the coverage of programming throughout the course, using weekly programming assignments tied to the material in the text.

Another option is to use the book as the basis for a fast-paced one-semester course. This choice is appropriate in situations where students in the sciences and engineering may have only one semester to devote to computer science. The course is best positioned early in the curriculum, because students who take it are sure to be able to use computers much more effectively when necessary in courses in their specialty. If the course is later in the curriculum, students may have a more mature point of view that better enables understanding of the advanced topics (but not much patience with learning the basics of programming).

Simply put, this book provides the following alternative to traditional introductory courses: a way to introduce students to computer science while at the same time teaching them to program, in a single course.

Prerequisites

Our aim is for the book to be suitable for typical first-year science and engineering students. Accordingly, we do not expect preparation significantly different from other entry-level science and mathematics courses.

Learning to program is one of our primary goals, so we assume no prior programming experience. Indeed, one of the most important features of our approach is that we integrate teaching programming with teaching the rest of computer science. But what should be done with students who have already taken a programming course, in high school or in college? Actually the material in this book is ideal for such students. They can review the mate-

rial on programming (which is likely to be in a different context from what they learned) and focus on the intellectual issues at the heart of the book (which are not likely to have been covered at all in their earlier course).

There is no substitute for experience when learning to program. Anyone writing a program to solve a new problem faces a challenging intellectual task, just as does anyone writing an essay on a new topic. Students who have written numerous essays in high school still benefit from introductory writing courses in college; just as students who have written numerous programs in high school can benefit from taking an introductory programming course. This analogy breaks down slightly because no one comes to college never having written an essay, but many students come to college never having written a program. But our experience has been that we can get students to the point where they can confidently write a program relatively quickly and that virtually all students in science and engineering can benefit from taking this as a first course (perhaps classified according to their programming experience).

Mathematical maturity is just as important. While we do not dwell on mathematical material, we do refer to the mathematics curriculum that students have taken in high school, including algebra, geometry, and trigonometry. Most students in our target audience (those intending to major in the sciences and engineering) automatically meet these requirements. *Discrete mathematics* plays a critical role in computer science but is not always fully covered in mathematics curricula, so we cover topics such as Boolean logic, mathematical induction, basic probability, and discrete sums with the point of view that most students have some familiarity with them but that the major concepts need to be reinforced (and we include an appendix with some basic information on discrete mathematics).

We also describe numerous *scientific applications,* integrated throughout the text. We occasionally draw on examples from probability, statistics, and calculus and Chapter 9, on scientific computing, covers several advanced topics. This material can be skipped by students who do not have requisite preparation (though they may wish to refer to it when they do take the more advanced courses). Otherwise, our examples do not assume any preparation beyond that provided by typical high-school courses in mathematics, physics, biology, or chemistry.

Goals

What can instructors of upper-level courses in science and engineering expect of students who have successfully completed a course based on this book?

Anyone who has taught an introductory computer science course knows that expectations are typically high: each instructor expects all students to be familiar with the computing environment and approach that he or she wants to use. A physics professor might expect some students to learn to program over the weekend to run a simulation; an engineering professor might expect other students to be using a particular package to numerically solve differential equations; or a computer science professor might expect knowledge of the details of a particular programming environment. Is it realistic to expect to be able to meet such diverse expectations? Should there be a different introductory course for each set of students? Colleges and universities have been wrestling with such questions since computers came into widespread use in the latter part of the 20th century.

Our primary goal is to provide an answer to such questions by developing a common introductory course in computer science for all students in science and engineering (including prospective computer science majors) that is analogous to commonly-accepted introductory courses in mathematics, physics, biology, and chemistry. The course may also be suitable for students in the humanities who wish a full introduction to the field.

Students who master the material in this book gain the confidence and knowledge that they need to be able to learn to exploit computers wherever they might appear later in their careers, whether it be using an integrated mathematical software package to attack advanced mathematical problems, using Java to develop innovative applications, writing code to control sophisticated devices, using simulation to study complex problems, or developing new computational paradigms. People continually need to be able to develop new skills in particular contexts because computers continue to evolve. Instructors teaching students who have studied from this book can expect that they have the background and experience necessary to make it possible for them to acquire such skills, to effectively exploit computers in diverse applications, and to appreciate their limitations.

Booksite

An extensive amount of information that supplements this text may be found on the world-wide web at

```
http://www.cs.princeton.edu/IntroCS
```

For economy, we refer to this web site as the *booksite* throughout. It contains material oriented towards instructors, students, and casual readers of the book. We briefly describe this material here, though, as all web users know, it is best surveyed by browsing. With a few exceptions to support testing, the material is all publicly available.

One of the most important implications of the booksite is that it empowers instructors and students to use their own computers to teach and learn the material in this course. Anyone with a computer and a browser can begin learning computer science and learning to program, by following a few instructions on the booksite. The process is no more difficult than downloading a media player or a new computer game.

For *instructors,* the booksite contains information about teaching the course. This information is primarily organized around a teaching style that we have developed over the past decade, where we offer two lectures per week to a large audience, supplemented by two class sessions per week where students meet in small groups with instructors or teaching assistants. The booksite has presentation slides for the lectures, which set the tone for the course.

We assign weekly problem sets based on exercises from the book and programming assignments, also based on exercises from the book, but with much more detail. Each programming assignment is intended to teach a relevant concept in the context of an interesting application while presenting an ambitious and interesting challenge to each student. The progression of assignments embody our approach to teaching programming. The booksite fully specifies all the assignments and provides detailed, structured information to support teaching students to complete them in the alloted time, including code for strawman solutions, descriptions of suggested approaches, and outlines for what should be taught in class sessions.

The booksite also includes webware for use in managing student submissions and grading assignments.

For *students*, the booksite contains quick access to much of the material in the book, plus extra material to encourage self-learning. Solutions are provided for many of the book's exercises, including complete program code and text data. There is a wealth of information associated with programming assignments, including suggested approaches, checklists, FAQs, and test data.

For *casual readers* (including instructors and students!) the booksite is a resource for accessing all manner of extra information associated with the book's content. All of the booksite content provides web links and other routes to pursue to find more information about the topic under consideration. There is far more information accessible than any individual could fully digest, but our goal is to provide enough to whet any reader's appetite for more information about the book's content.

As with any web site, our *Introduction to Computer Science* booksite is continually evolving, but it is an essential resource for everyone who owns this book. In particular, the supplemental materials supporting teaching and learning within a first-year college course are critical to our goal of making computer science an integral component of the education of all scientists and engineers.

Acknowledgements

Contents

1 *Overview*

2 *Elements of Programming*

Our goal in this chapter is to convince you that writing a program is easier than writing a piece of text such as a paragraph or an essay. Writing prose is difficult: we spend many years in school to learn how to do it. By contrast, just a few building blocks suffice to take us into a world where we can harness the computer to help us solve all sorts of fascinating problems that would be otherwise unapproachable. In this chapter, we take you through these building blocks, get you started on programming in Java, and study a variety of interesting programs. You will have an additional avenue to be able to express yourself (writing programs) within just a few weeks. Like the ability to write prose, the ability to program is a lifetime skill that you can continually refine and is certain to serve you well into the future.

You will learn the Java programming language, but that will be much easier to do than learning a foreign language that is unfamiliar to you. Indeed, programming languages are characterized by no more than a few dozen vocabulary words and rules of grammar. Most of the material that we cover in this chapter could apply to the C or C++ languages, or any of several other modern programming languages, but we describe everything specifically in Java so that you can get started creating and running programs right away. On the one hand, to the extent possible, we will focus on learning to program, as opposed to learning details about Java. On the other hand, part of the challenge of learning to program is knowing which details are relevant in a given situation. Java is widely available, but also learning to program in Java will make it easy for you learn to program in another language.

2.1 Your First Program

In this section, our plan is to lead you into world of Java programming by taking you through the basic steps required to get a simple program running. The Java system is a collection of applications not unlike any of the other applications that you are accustomed to using (such as your word processor, e-mail program, or internet browser). As with any application, you need to be sure that Java is properly installed on your computer. It comes preloaded on may computers and is easy to download. You also need an editor and a terminal application (see Appendix X).

Programming in Java. To introduce you to developing computer programs written in Java, we break the process down in to three steps. To program in Java you need to:

1. *create* the program by typing it into a file named, say, `MyProgram.java`
2. *compile* it by typing `javac MyProgram.java` in a terminal window
3. *run* (or *execute*) it by typing `java MyProgram`

In the first step, you start with a blank page and end with a sequence of typed characters on the page, just as when you write an e-mail or a paper. In the second step, you use a system application called a *compiler* that translates your program into a form more suitable for the computer (and puts the result in a file named `MyProgram.class`). In the third step, you transfer control of the computer from the system to your program (which returns control back to the system when finished).

Creating a program. A program is nothing more than a sequence of characters, like a sentence, a paragraph, or a poem. To create one, therefore, we need only define that sequence of characters, in the same way as we do for e-mail or any other computer application. Programmers usually use a simple editor for this task with a fixed-width font to make it easier to line things up vertically in the code.

Compiling a program. At first, it might seem to you as though the Java programming language is designed to be best understood by the computer. Actually, to the contrary, the language is designed to be best understood by

the programmer (that's you). The computer's language is more primitive than Java, as we shall see in Chapter 5. A *compiler* is an application that translates program from the Java language to a language more suitable for executing on the computer. It takes a file with a `.java` extension as input (your program) and produces a file with the same name but with a `.class` extension (the computer-language version). To use the compiler to compile a program, type the `javac` command followed by the program name in a terminal window. Most systems have other ways to interact with the compiler; we choose this one here because its use is simple and compact to describe.

Executing a program. Once it has been compiled, we can run the program. This is the exciting part, where your program actually takes control of your computer (within the constraints of what the Java system allows). It is perhaps more acurate to say that the computer follows your instructions.

Program 2.1.1 is an example of a complete Java program. Its name is `HelloWorld`, so that its code must reside in a file named `HelloWorld.java` (by convention in Java). The program's sole action is to print a message back to the terminal window. For continuity, we will use some standard Java terms to describe the program without defining them until Section X, but you do not now need to know details of these definitions: Program 2.1.1 consists of a single *class* named `HelloWorld` that has a single *method* named `main()` that uses a method named `println()` from Java's `System.out` library to do the job. for the time being, you can think of "class" as meaning "collection of programs" and "method" as meaning "program." When referring to methods, we use `()` after the name to distinguish method names from other kinds of names.

Since the 1970s, it has been a tradition that everyone learning to program should start with `HelloWorld`, so you should type it into a file, compile it, and run it. By doing so, you will not just be following in the footsteps of countless other people who have learned how to program, but you also will be checking that you have a usable editor and terminal emulator (see Appendix X) and that your Java system is properly installed.

A method comprises a *signature*, which has its name and other information, and a *block*, which is sequence of *statements* enclosed in braces and each followed by a semicolon. The statements in a method's block are exe-

Program 2.1.1 Hello, World

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

This code is a complete Java program that accomplishes a simple task (printing a message on the terminal window). It is traditionally a beginning programmers' first program .

```
% javac HelloWorld
% java HelloWorld
Hello, World
```

cuted, one by one, when the method is invoked. One type of statement is a method name with, in parentheses, zero or more arguments. When we write a such a statement, we are simply saying that we want to run that method (and to provide it with some information, in the arguments). This process is known as *invoking* or *calling* the method. In `HelloWorld`, the `main` method consists of a single statement that calls the Java library method `System.out.println()`. The name of the method is `println()`; the name of the library is `System.out`. You will be writing programs that use calls to many different Java library methods, and you refer to each of them in just this way. In Program 2.1.1, `System.out.println()` takes the message `Hello, World` as its argument and prints it to the terminal. We do not need to know the details of how `System.out.println()` accomplishes this task—we can simply rely on it to do so. The method `main()` itself takes an argument, which we will discuss soon as the focus of Program 2.1.2.

At first, accomplishing the task of printing something out in a terminal window might not seem very interesting; upon reflection, you will see that one of the most basic capabilities that we need from a program is for it to be able to tell us what it is doing.

For the time being, all our program code will be just like Program 2.1.1, except with a different sequence of statements in `main()`. We

will develop a more detailed understand of everything in `HelloWorld.java` in due time: for the moment, you do not need to start with a blank page to write a program, because you can

- copy `HelloWorld.java` into a new file whose name is a new program name of your choice, followed by `.java`
- replace `HelloWorld` with the program name on the first line
- replace the `System.out.println()` statement by a sequence of statements (each ending with a semicolon).

Each Java program must reside in a file whose name matches the name after the word `class` on the first line and has a `.java` extension. In the next two sections, you will learn all kinds of different statements that you can put together to make a program; for the moment, we will just use `System.out.println()` and its cousin `System.out.print()` (which does the same thing except without starting a new line after printing the message).

Errors. In modern interactive systems like Java, we sometimes blur the distinction among editing, compiling, and executing programs, but it is worthwhile to keep them separate in your mind when you are learning to program. The main reason to do so is to understand the effects of the errors that inevitably crop up. Several examples of errors are discussed in the Q&A and the exercises at the end of this section. Most errors are easily fixed by carefully examining the program as we create it, in just the same way as we fix spelling and grammatical errors when we type an e-mail message. Some other errors are caught when we compile the program, because they prevent the compiler from doing the translation (so it issues an error message that tries to explain why). Many other errors, called *bugs*, do not show up until we execute the program. Errors are the bane of a programmer's existence: the compiler's error messages can be confusing or misleading, and bugs can be very hard to find. One of the very first skills that you will learn is to identify errors; one of the next will be to be sufficiently careful when coding to avoid many of them.

Program 2.1.2 uses three statements to accomplish a more complicated task than simply printing a message. Whenever this program is executed, it reads the *command-line argument* that you type after the program name and prints it back out to the terminal as part of the message. The result of executing this program depends on what we type after the program name.

Program 2.1.2 Using a command-line argument

```
public class Hi
  {
    public static void main(String[] args)
      {
        System.out.print("Hi, ");
        System.out.print(args[0]);
        System.out.println(". How are you?");
      }
  }
```

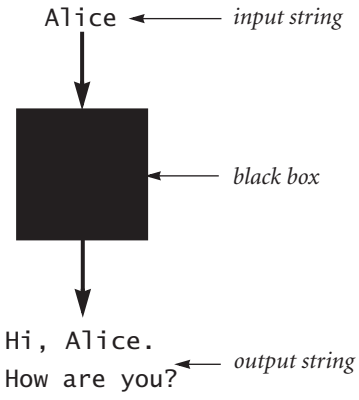
This program shows the way in which we can control the actions of our programs: by providing an argument on the command line. Doing so allows us to tailor the behavior of our programs.

```
% javac Hi.java
% java Hi Alice
Hi, Alice. How are you?
% java Hi Bob
Hi, Bob. How are you?
```

After compiling the program once, we can run it for different command-line arguments and get different results.

As with your first program, we will discuss in more detail the mechanism that we use to pass arguments to our programs later, in Section X. In the meantime, you can do so by using `args[0]` to represent the argument, just as in Program 2.1.2.

Again, accomplishing the task of getting a program to write back out what we type in to it may not seem interesting at first, but upon reflection you will see that another basic capability that we need from a program is for it to be able to respond to basic information from the user to control what it does. The simple model that Program 2.1.2 represents will suffice to allow us to consider Java's basic programming mechanism and to address all sorts of interesting computational problems. Later, we will consider refinements to this simple model that incorporate more sophisticated mechanisms for program input and output.



A bird's-eye view of a Java program

Stepping back, we can see that Program 2.1.2 does no more nor no less than implement a *function* that maps a string of characters (the argument) into another string of characters (the message printed back to the terminal). When using it, we might think of our Java program as a black box that converts our input string to some output string. This model is attractive because it is very simple, but also sufficiently general to allow completion of any computational task. For example, the Java compiler itself is nothing more than a program that takes one string of characters as input (a `.java` file) and produces another string of characters as output (the corresponding `.class` file). While we stop short of programs quite that complicated, we will be able to consider and to write programs that accomplish a variety of interesting tasks.

Q&A

Q Why Java?

A The programs that we are writing are very similar to their counterparts in several other languages, so our choice of language is not crucial.. We use Java because it is widely available, embraces a full set of modern abstractions, and has a variety of automatic checks for mistakes in programs, so it is suitable for learning to program. There is no perfect language, and you certainly will find yourself programming in other languages in the future.

Q What are Java's rules regarding tabs, spaces and newline characters?

A There are not many. Java translators treat them all to be equivalent. For example, we could also write Program 2.1.1 as follows:

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

But we do normally adhere to spacing and indenting conventions when we write Java programs, just as we always indent paragraphs and lines consistently when we write prose or poetry.

Q What are the rules regarding quotation marks?

A Material inside quotation marks is an exception to the rule of the previous question: things within quotes have to be taken literally so that you can precisely specify what gets printed out. If you put any number of successive spaces within the quotes, you get that number of spaces in the output. If you accidentally omit a quotation mark, the compiler may get very confused, because it needs that mark to know the difference between characters in the string you are defining and you program itself.

Q How do I print out a quotation mark, a newline, or a tab?

A Use `\"`, `\n`, or `\t`, respectively, within the quotation marks.

Q What happens when you omit a brace or misspell one of the words, like `public` or `static` or `main`?

A It depends upon precisely what you do. Such errors are called *syntax errors* and are usually caught by the compiler. But the compiler may not be able to tell you exactly what mistake you made (for a profound reason that we will discuss in Chapter 8), so it might print an error message that is hard to understand. For example, if you make a program `Bad.java` that is exactly the same as Program 2.1.1 except that you omit the first left brace (and change the program name from `HelloWorld` to `Bad`), you get the following helpful message:

```
% javac Bad.java
Bad.java:1: '{' expected
public class Bad
                ^
1 error
```

From this message, you might correctly surmise that you need to insert a left brace. But if you omit only the second left brace, you get a less helpful sequence of messages:

```
Bad.java:3: ';' expected
    public static void main(String[] args)
                                           ^
Bad.java:6: 'class' or 'interface' expected
  }
  ^
Bad.java:7: 'class' or 'interface' expected

^
Bad.java:3: missing method body, or declare abstract
    public static void main(String[] args)
                        ^
4 errors
```

One way to get used to such messages is to intentionally introduce mistakes into a simple program, then see what happens. Whatever the error message says, you should treat the compiler as a friend, for it is just trying to tell you that something is wrong with your program.

Q Can a program use more than one command-line argument?

A Yes, you can put several, though we normally use only a few. You refer to the second one as `args[1]`, the third one as `args[2]`, and so forth.

Q Do I really have to type in the programs in the book to try them out?

A No, you can find them on the web site

```
http://www.cs.princeton.edu/IntroCS
```

which we refer to as the *booksite*. This site also has information about installing and running Java on your computer, answers to some exercises, web links, and other extra information that you might find useful or interesting. For example, the booksite describes how to annotate code with *comments*, by enclosing the comment in between `/*` and `*/` or by putting the comment at the end of any line after `//`. We do not use comments in the book because the text describes the code.

Exercises

2.1.1 Write a program that prints the `Hello, World` message 10 times.

2.1.2 Describe what happens if, in `HelloWorld.java`, you omit

a `public`

b `static`

c `void`

d `args`

2.1.3 Describe what happens if, in `HelloWorld.java`, you misspell (by, say, omitting the second letter)

a `public`

b `static`

c `void`

d `args`

2.1.4 Describe what happens if you try to execute Program 2.1.2 with

a `java Hi`

b `java Hi @!&^%`

c `java Hi 1234`

d `java Hi.class Bob`

e `java Hi.java Bob`

f `java Hi Alice Bob`

2.1.5 Modify `Hi.java` to make a program `HiThree.java` that takes three names and prints out a proper sentence with the names in the reverse of the order given, so that, for example, `java HiThree Alice Bob Carol` gives `Hi Carol, Bob, and Alice`.

2.1 Primitive Types of Data

A *data type* is a set of values and a set of operations defined on them. For example, we are familiar with numbers and with operations defined on them such as addition and multiplication. In mathematics, we are accustomed to thinking of the set of numbers as being infinite; in computer programs we have to work with a finite number of possibilities. In Java, you must always be aware of the type of the data that your program is processing.

There are eight different built-in types of data (called *primitive* types) in Java, mostly for different kinds of numbers. Of the eight primitive types, we most often use three: `int` for integers; `double` for real numbers; and `boolean` for true-false values. There are other types of data (called *system* types) available in Java libraries: for example, the only type used in the programs in Section 2.1 is the system type `String` for strings of characters. We consider the `String` type in this section because its usage for input and output is essential. Accordingly, it shares some characteristics of the primitive types: for example, some of its operations are built in to the Java language. Beyond primitive types and system types, we can build our own data types: indeed, programming in Java is often centered around doing so, as we shall see in Chapter 3.

After defining some basic terms, we will consider several sample programs that illustrate use of different types of data. These programs still do not do much real computing, but understanding types is an essential step in beginning to program and sets the stage for us to begin working with more intricate programs in the next section.

Basic definitions. To talk about data types, we need to define a number of terms, which are illustrated in the following simple four-statement Java program fragment:

```
int a, b, c;
a = 1234;
b = 99;
c = a + b;
```

The first statement is a *declaration* that declares three *variables* with the *identifiers* a, b, and c to be of type `int`. The next three statements are *assignment*

statements that change the values of the variables, using the *literals* 1234 and 99, with the end result that `c` has the value 1243.

Identifiers. We use identifiers to name variables (and many other things) in Java. An identifier is a sequence of letters, digits, _ and `$`, the first of which is not a digit. The sequences of characters `abc`, `Ab$`, `abc123`, and `a_b` are all legal Java identifiers, but `Ab*`, `1abc`, and `a+b` are not.

Literals. For each of the primitive data types, Java defines ways to specify its values. To specify an integer, we use a string of digits like `1234` or `99` to define `int` literal values. To specify a real number, we add a decimal point as in `3.14159` or `2.71828` to define `double` literal values. To specify a `boolean` value, we use the keywords `true` or `false`. We will consider other literals as we consider each type in more detail.

Variables. A variable is an instance of a data type. We create one by declaring its type and giving it a name; we compute with it by using the name in an expression that uses operations defined for its type. Each variable has one of the data-type values.

Declarations. We always use declarations to specify the names and types of variables in our programs. By doing so, we are being explicit about any computation that we are specifying. The Java compiler checks for consistency (for example, it does not make sense to multiply a `String` by a `double`) and translates to the appropriate machine code.

Assignment statements. When we write `a = b + c` in Java, we are not expressing mathematical equality, but are instead expressing an action: set the value of the variable `a` to be the value of `b` plus the value of `c`. It is true that `a` is mathematically equal to `b + c` after the assignment statement has been executed, but the point of the statement is to change the value of `a` (if necessary). The left-hand side of an assignment statement must be a single variable (identifier); the right-hand side can be an arbitrary expression that uses operators defined for the type. For example, we can say `discriminant = b*b - 4*a*c` in Java, but we do not say `a + b = b + a` or `1 = a` , and `a = b` is certainly not the same as `b = a`.

Expressions. We can use parentheses to specify compound operations to compute a value just as in mathematical formulas. For example, we can write `(x + 1) * (x + 1)` or `x*x + 2*x + 1` on the right-hand side of an assignment statement and the compiler will understand what we mean. An expression is shorthand for specifying a sequence of computations: in what order should they be performed? Java has natural and well-defined *precedence* rules (see Appendix X) that fully specify this order. For arithmetic operations, multiplication and division are performed before addition and subtraction. When the precedence rules do not lead to the computation that you want, you can use parentheses.

Functions. Some methods implement functions—they use their arguments to compute a value of a specified type. Besides variables and literals, we can use method names (with arguments) in expressions, when we do so, we are expecting that method to compute a value of the appropriate type. Method arguments may also be expressions. Many familiar functions are implemented in Java's `Math` libarary (see Appendix X). Thus, we can write expressions like `Math.sin(x)*Math.cos(y)` and so on.

For economy, we can combine a declaration with an assignment statement to provide an initial value for the variable (otherwise, the value is undefined). Declarations can appear anywhere in a program before a variable is used—most often, we put them at the point of first use, as in this version of the program fragment that we started with.

```
int a = 1234;
int b = 99;
int c = a + b;
```

In older programming languages, it was required that all declarations appear at the beginning of a piece of code, and it is legal to do so in Java. We sometimes use this convention in small programs with multiple variables (for an example, see Program 2.1.2), but most often we put declarations at the point of first use.

To understand how to use a primitive data type, you need to know its defined set of values, which operations you can perform, and how to specify literals. Next, we consider these details for characters, strings, integers, float-

ing point numbers, and true–false values, along with sample programs that illustrate their use.

Characters and Strings. A `char` is an alphanumeric character or symbol, like the ones that you type. There are 2^{16} different possible character values, but we usually restrict attention to the ones that represent letters, numbers, symbols, and white-space characters like tab and newline. To specify a character as a `char` literal, we enclose it in single quotes: `'a'` represents the letter a. For tab, newline, single quote and double quote, we use the special *escape sequences* `\t`, `\n`, `\'`, and `\"`, respectively. The characters are encoded as 16-bit integers using an encoding scheme known as Unicode, and there are escape sequences for specifying any special character (see Appendix X). We usually do not perform any operations on characters other than assigning values to variables and using the comparison functions defined later in this section, so we move immediately on to other data types.

A `String` is a sequence of characters. A literal `String` is a sequence of characters within double quotes, like `"Hello, World"`. The `String` data type is *not* a primitive type, but Jave sometimes treats it like one. For example, one of the most common operations that we perform on `String`s is built in to the Java language. That operation is known as *concatenation*: given two strings, chain them together to make a new string. The symbol for concatenating strings in Java is + (the same symbol that we use for adding numbers). For example, the following Java code fragment is like the one that we considered at the beginning of this section, but it uses `String`s instead of `int`s:

```
String a, b, c;
a = "1234";
b = "99";
c = a + b;
```

As before, the first statement declares three variables to be of type `String` and the next three assign values to them. In this case, the end result that `c` has the value `"123499"`.

We use the concatenation operation frequently to put together results of computation in output. For example we could make Program 2.1.2 much simpler by replacing its three statements by this single statement:

Program 2.1.1 String-concatenation example

```
public class Ruler
  {
    public static void main(String[] args)
      {
        String ruler1 = "1 ";
        String ruler2 = ruler1 + "2 " + ruler1;
        String ruler3 = ruler2 + "3 " + ruler2;
        String ruler4 = ruler3 + "4 " + ruler3;
        System.out.println(ruler1);
        System.out.println(ruler2);
        System.out.println(ruler3);
        System.out.println(ruler4);
   }

}
```

This program prints the relative lengths of the subdivisions on a ruler, as shown, for example, by the lines at right for the third line. The length of the output of this program grows exponentially: the nth line has $2^n - 1$ *numbers on it.*

```
% javac Ruler
% java Ruler
1
1 2 1
1 2 1 3 1 2 1
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

```
System.out.println("Hi, " + args[0] + ". How are you?");
```

The concatenation operation (along with the ability to declare `String` variables and to use them in expressions and assignment statements) is sufficiently powerful to allow us to attack some nontrivial computing tasks, as illustrated in Program 2.1.1 and Exercise 2.1.18.

How do we print out the values of other types of data? To print out anything, we need to convert it from its internal representation to a `String`. Similarly, when we put a value on the command line, we type a string of characters: if we want to process the data that the string represents, we must convert the string to the appropriate internal reservation for that type of data.

Converting from one type of data to another is an important issue that we often need to face when writing programs; we will discuss it at length at the end of this section. We are discussing type conversion to and from `String` briefly now so that we can write sample programs that take command-line arguments and print out values for various types of data.

To convert from one type of data to another, we might explicitly call a system method or we might expect the system to automatically do the conversion. In Java, we normally use the former approach for input from the command line and the latter for output.

For input from the command line, we explicitly use system methods to convert from `String` to the appropriate type. For example, the system method `Integer.parseInt` takes a string parameter and converts the given string of digits to an `int`. For output, we take advantage of the concatenation operator and Java's *automatic type conversion* facility to write statements like this one:

```
System.out.println(a + " + " + b + " = " + (a+b));
```

If `a` and `b` are `int` variables with the values `1234` and `99`, respectively, then this statement prints out the string `1234 + 99 = 1243`. This usage takes advantage of the Java convention that if one of the operands of + is a `String`, then the other is automatically converted to a `String` so that the + is interpreted as concatenation. We will discuss the specific rules for such conversions in more detail at the end of this section, after we have discussed properties of each of the types.

Integers. An `int` is an integer (whole number) between -2147483648 and 2147483647. These bounds derive from the fact that integers are represented in binary with with 32 binary digits: there are 2^{32} possible values. (The term *binary digit* is omnipresent in computer science, so the abbreviation *bit* is always used: a bit is either 0 or 1.) The range of possible `int` values is asymmetric because 0 is included with the positive values. We will consider number representations in detail in Chapter 5 (see also Appendix X); in the present context it suffices to know that an `int` is one of the finite set of values in the range just given. We use `int`s frequently not just because they occur

Program 2.1.2 Integer multiplication and division

```
public class IntOps
  {
    public static void main(String[] args)
      {
        int a, b, p, q, r;
        a = Integer.parseInt(args[0]);
        b = Integer.parseInt(args[1]);
        p = a * b;
        q = a / b;
        r = a % b;
        System.out.println(a + " * " + b + " = " + p);
        System.out.println(a + " / " + b + " = " + q);
        System.out.println(a + " % " + b + " = " + r);
        System.out.print(a + " = ");
        System.out.println(q + " * " + b + " + " + r);
      }
}
```

Arithmetic on integers is built in to Java. Most of this code is devoted to the tasks of getting the values in and out; the actual arithmetic is in the simple statements like `p = a * b`.

```
% javac IntOps
% java IntOps 1234 99
1234 * 99 = 122166
1234 / 99 = 12
1234 % 99 = 46
1234 = 12*99 + 46
```

frequently in the real world, but also they naturally arise when expressing algorithms.

Standard arithmetic operators for addition/subtraction (`+` and `-`), multiplication (`*`), division (`/`), and remainder (`%`) for the `int` data type are built in to Java. These operators take two `int` arguments and produce an `int` result, with one significant exception: division or reminder by 0 is not allowed. These operations are defined just as in grade school: Given two `int`s `a` and `b`, the value of `a / b` is the number of times `b` goes into `a` and the value of `a % b` is the remainder that you get when you divide `a` by `b`. The `int` result that we get from these operations is just what we expect from the grade-school definition, with one significant exception: if the result is too large to fit into the 32-bit representation described above, then it will be truncated to the

least significant 32 bits. We have to take care that such a result is not misinterpreted by our code.

Integer literals are just strings of decimal integers, as in the code fragment above. To use integer parameters on the command line, we need to use the built-in Java method `Integer.parseInt`, which converts string values to integer values. When we type a number as a command-line parameter, it is a string of characters, so we have to include a statements that calls with `Integer.parseInt` to convert it to an integer at execution time; if we type the same string of characters within the program, the compiler interprets it as a literal integer and does the conversion at compile time. Program 2.1.2 illustrates these basic operations for manipulating integers.

Three other built-in types are different representations of integers. The `long`, `short`, and `byte` types are the same as `int` except that they use 64, 16, and 8 bits respectively, so the range of allowed values is accordingly different. Programmers use `long` when working with huge integers and the others to save space. Appendix X gives a table with the maximum and minimum values for each type.

Real numbers. The `double` type is for representing *floating-point* numbers, for use in scientific and commercial applications. The internal representation is like scientific notation, so that we can compute with numbers in a huge range. Floating-point numbers are not the same as real numbers, primarily because there are an infinite number of real numbers, and we can only represent finitely many numbers in *any* digital computer representation. Floating-point numbers do approximate real numbers sufficiently well that we can use them in all sorts of applications, but we often need to cope with the fact that we cannot always do exact computations.

We can use a string of digits with a decimal point to type floating-point numbers. For example, `3.14159` represents a six-digit approximation to π. Alternatively, we can use a notation like scientific notation: the literal `6.022E23` represents the number 6.022×10^{23}. As with integers, you can use these conventions to write floating-point literals in your programs or to provide floating-point numbers as string parameters on the command line.

The arithmetic operators `+`, `-`, `*`, and `/` are defined for `double`. The result of a calculation can be `0` (if the number is too small to be represented)

Program 2.1.3 Quadratic formula

```
public class Quadratic
  {
    public static void main(String[] args)
      {
        double b = Double.parseDouble(args[0]);
        double c = Double.parseDouble(args[1]);
        double discriminant =  Math.sqrt(b*b - 4.0*c);
        System.out.println((-b + discriminant) / 2.0);
        System.out.println((-b - discriminant) / 2.0);
      }
  }
```

This program prints the roots of the polynomial $x^2 + bx + c$, using the quadratic formula. For example, the roots of $x^2 - 3x + 2$ are 1 and 2 since we can factor it as $(x-1)(x-2)$; the roots of $x^2 - x - 1$ are φ and $1-\varphi$, where φ is the golden ratio; and the roots of $x^2 + x + 1$ are not real numbers.

```
% javac Quadratic
% java Quadratic -3.0 2.0
2.0
1.0
% java Quadratic -1.0 -1.0
1.618033988749895
-0.6180339887498949
% java Quadratic 1.0 1.0
NaN
NaN
```

or one of the special values `Infinity` (if the number is too large to be represented) or `NaN` (if the result of the calculation is undefined). Beyond the built-in operators, the Java `Math` library defines the square root, trigonometric functions, logarithm/exponential functions and other common functions for floating-point numbers (see Appendix X).

There are myriad details to consider with regard to precision and when calculations involve `0`, infinity or `NaN` (see Appendix X and the Q&A and exercises at the end of this section) but you can use these operations in a natural way for all sorts of familiar calculations. With this library, you can begin to write Java programs instead of using a calculator for all kinds of calculations. For example, Program 2.1.3 shows the use of `double`s in computing roots of a quadratic equation using the quadratic formula. Several of the exercises at the end of this section further illustrate this point.

As with `short` and `byte` for integers, there is another representation for real numbers, called `float`, that programmers sometimes use to save space when precision is a secondary consideration. The `double` type is useful for about 15 significant digits; the `float` type is good for only about 7 digits. We rarely use `float` in this book.

Booleans. The `boolean` type has just two values: `true` or `false`. These are the two possible `boolean` literals, every `boolean` variable has one of these two values, and every boolean operation has operands and a result that takes on just one of these two values. This apparent simplicity is deceiving—booleans lie at the foundation of computer science, as we will see throughout this book.

The most important operators defined for `boolean`s are for the *and* (`&&`), *or* (`||`), and *not* (`!`) functions, which are listed in the table below along with two different descriptions of each of their meanings:

and	`&&`	`a && b` is `true` if both `a` and `b` are `true` `a && b` is `false` if either `a` or `b` are `false`
or	`||`	`a || b` is `true` if either `a` or `b` are `true` `a || b` is `false` if both `a` and `b` are `false`
not	`!`	`!a` is `true` if `a` is `false` `!a` is `false` if `a` is `true`

Although these definitions are intuitive and easy to understand, it is worthwhile to fully specify each possibility for each operation, as in the following tables. Such tables are known as *truth tables*.

a	!a
true	false
false	true

a	b	a && b	a || b
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

The *not* function (on the left) has only one operand: its value for each of the two possible values of the operand is specified in the second column. The *and*

Program 2.1.4 Computing with boolean values

```
public class LeapYear
  {
    public static void main(String[] args)
      {
        boolean isLeapYear;
        int year = Integer.parseInt(args[0]);
        isLeapYear = year % 4 == 0;
        isLeapYear = isLeapYear && (year % 100 != 0);
        isLeapYear = isLeapYear || (year % 400 == 0);
        System.out.println(isLeapYear);
      }
  }
```

This program tests whether an integer corresponds to a leap year in the Gregorian calendar. A year is a leap year if it is divisible by 4 (2004) unless it is divisible by 100 in which case it is not (1900), unless it is divisible by 400 in which case it is (2000).

```
% javac LeapYear
% java LeapYear 2004
true
% java LeapYear 1900
false
% java LeapYear 2000
true
```

and *or* functions each have two operands: there are four different possibilities for operand input values, and the values of the functions for each possibility are specified in the right two columns.

We can use these operators with parentheses to develop arbitrarily complex expressions, each of which specifies a well-defined boolean function. Often, the same function appears in different guises. For example, the expressions `(a && b)` and `!(!a || !b)` are equivalent.

The study of manipulating expressions of this kind is known as *Boolean logic* and is the subject of Section 5.1. This field of mathematics is fundamental to computing. For example, Boolean logic plays an essential role in the design and operation of computer hardware itself, as we will see in Chapter 5. It also helps us understand fundamental limitations on what we can compute, as we will see in Chapter 8.

In the present context, we are interested in boolean expressions because we use them to control the behavior of our programs. Typically, a particular condition of interest is specified as a boolean variable and a piece of program code is written to execute one set of statements if the variable is `true` and a different set of statements if the variable is `false`. The mechanics of doing so are the topic of Section X.

Comparisons. Some *mixed-type* operations take operands of one type and produce a result of another type. The most important built-in operations of this kind are the *comparison* operations ==, !=, <, <=, >, and >=, which all are defined for each primitive numeric type and which all produce a `boolean` result. As we will see in Section X, these operations play a critical role in the process of developing programs more sophisticated than the sequence-of-statements programs that we have been considering. The meanings of these operations are summarized in the following table:

op		a *op* b	true	false
==	*equal*	a is equal to b	2 == 2	2 == 3
!=	*not equal*	!(a == b)	3 != 2	2 != 2
<	*less than*	a is less than b	2 < 13	2 < 2
<=	*less than or equal*	(a < b) || (a == b)	2 <= 2	3 <= 2
>	*greater than*	!(a <= b)	13 > 2	2 > 13
>=	*greater than or equal*	!(a < b)	3 >= 2	2 >= 3

Since operations are defined only with respect to data types, each of these symbols stands for many operations, one for each data type. It is required that both operands be of the same type, and the result is always `boolean`. Even without going into the details of number representation that we consider in Section X, it is clear, on reflection, that the operations for the various types are really quite different: for example, it is one thing to compare two `int`s to check that `2 == 2` is `true` but quite another to compare two `double`s to check whether `2.1 == .0021E3` is `true` or `false`.

Comparison operations, with boolean logic, provide the basis for decision-making in Java programs. Program 2.1.4 is a simple example of their use, and you can find other examples in the exercises at the end of this section. In Section 2.2 we will see the critical role that such expressions play in controlling the flow of execution of the statements in our programs.

Type conversion. One of the first rules of programming in Java is that you should always be aware of the type of data that your program is processing. Only by knowing the type can you know precisely what set of values each variable can have, what literals you can use, and what operations you can perform. But typical programming tasks involve processing multiple types of data, and we often need to be able to convert data from one type to another. There are several ways to do so in Java.

Explicit type conversion. You can use a method that takes an argument of one type (the value to be converted) and returns a result of another type. For example, the library method `Math.round` takes a `double` argument and returns an `long` result: the nearest integer to the argument. Thus, for example, `Math.round(3.14159)` and `Math.round(2.71828)` are both of type `long` and have the same value (`3`). Our use of the library methods `Integer.parseInt` and `Double.parseDouble` in the programs in this section are also examples of this kind of conversion. In Section 2.4, you will learn how to implement such methods.

Automatic conversion for numbers. You can use data of any primitive numeric type where a value whose type has a larger range of values is expected. Java automatically converts to the type with the larger range. For example, we used numbers all of type `double` in Program 2.1.3, so there is no conversion, but we might instead have chosen to make `b` and `c` of type `int` (using `Integer.parseInt` to convert the command-line arguments). Then the expression `b*b-4.0*c` requires type conversion, which Java would do automatically: first, it would covert `c` to `double` and multiply by the `double` literal `4.0` to get a `double` result. As for the other term, it would compute the value `b*b` and then convert to `double` for the subtraction, leaving a `double` result. Or, we might have written `b*b - 4*c`. In that case, the program would evaluate the expression `b*b - 4*c` as an `int` and then automatically convert

Program 2.1.5 Casting to get a random integer

```
public class RandomInt
  {
    public static void main(String[] args)
      {
        int N = Integer.parseInt(args[0]);
        double r = Math.random();
        int n = (int) (r * N);
        System.out.println(n);
      }
  }
```

This program uses the Java method `Math.random` *to get a random number between 0 and 1, then multiplies it by the command-line argument N to get a random number between 0 and N, then uses a cast to truncate the result to be an integer between 0 and N-1.*

```
% javac RandomInt
% java RandomInt 1000
548
% java RandomInt 1000
141
% java RandomInt 1000000
135032
```

the result to `double`, because that is what `Math.sqrt` expects. Java does this kind of type conversion automatically because your intent is clear and it can do so with no loss of information. On the other hand, if the conversion might involve loss of information, for example if you try to assign a `double` to an `int`, you will get an error message from the compiler.

Explicit cast. Java has some built-in type conversion methods for primitive types that you can use when you are aware that you might lose information, but you have to make your intention to do so explicit using a device called a *cast*. You cast a variable or an expression from one primitive type to another simply by prepending the desired type name within parentheses: for example, the expression `(int) 3.14159` is a cast from `double` to `int` that is of type `int` with value `3`. Java has built-in conversion methods defined for casts among primitive numeric types that implement reasonable and convenient approaches to throwing away information (for a full list, see Appendix X).

One that we use frequently is to cast a floating-point number to an integer: the built-in method throws away the fractional part by rounding towards zero. If you want a different result, such as rounding to the nearest integer, you must use the explicit conversion method `Math.round`, as just discussed (but you then need to use an explicit cast to `int`, since that method returns a `long`). Program 2.1.5 is an example that uses a cast for a practical computation.

Automatic conversion for strings. As mentioned at the beginning of this section, the Java built-in `String` type obeys special rules. One of these special rules is that you can easily convert any type of data to a `String`: Java invokes the appropriate method automatically whenever we use the + operator with one of its operands a `String`, producing as a result the concatenation of the `String` operand and the result of converting the non-`String` operand to a `String`, in the order given. For example, the result of the code fragment

```
String a = "";
int b = 99;
String c = a + b;
```

is that `c` has the value `"99"`. We use this automatic conversion liberally in our programs: see Program 2.1.2 for an example.

Summary. A data type is a set of values and a set of operations on those values. Java has eight primitive data types: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. The `boolean` type is for computing with the logical values `true` and `false`; the `char` type is the set of character values that we type; and the other six are numeric types, for computing with numbers. Another data type, `String`, is not primitive, but Java has some built-in facilities for `String`s that are like those for primitive types.

When programming in Java, we have to be aware that all operations must be from the type of their operands (so we may need type conversions) and that all types can have only a finite number of values (so we may need to live with imprecise results).

The numeric types and Java's libraries give use the ability to use Java as an extensive mathematical calculator: we can use Java's `Integer.parseInt` and `Double.parseDouble` methods to convert command-line arguments to

numeric values for primitive types, write expressions assigning values to variables using +, -, *, /, and % and Java methods from the Math library, and then use automatic type conversion to String with the + operator and Java's System.out.println method to output results. Each Java program is still a black box that takes string arguments and produces string results, but we can now interpret those strings as numbers and use them as the basis for meaningful computation.

The boolean type and its operations &&, ||, and ! are the basis for logical decision-making in Java programs, when used in conjunction with the mixed-type comparison operators ==, !=, <, >, <=, and >=.

While the programs of this section are quite primitive by the standards of what we will be able to do after the next section, this class of programs is quite useful in its own right. You will use primitive types extensively in Java programming, so the effort that you spend now understanding them will certainly be worthwhile.

Q&A

Q What is the difference between = and ==?

A Your ability to answer this question is a sure test whether or not you understood the material in this section. Read it again carefully to find the answer, and then think about how you might explain the difference (succinctly) to a friend.

Q What happens if you forget to declare a variable?

A The compiler complains, as shown below for a program IntOpsBad that is the same as Program 2.1.2 except that the variable p is omitted from the declaration statement.

```
javac IntOpsBad.java
IntOpsBad.java:7: cannot resolve symbol
symbol  : variable p
location: class IntOpsBad
        p = a * b;
```

```
        ^
IntOpsBad.java:10: cannot resolve symbol
symbol  : variable p
location: class IntOpsBad
        System.out.println(a + " * " + b + " = " + p);
                                                   ^
2 errors
```

The compiler says that there are two errors, but there is really just one: the declaration of `p` is missing. If you forget to declare a variable that you use often, you may get quite a few error messages.

Q Why do `int` values sometimes turn into negative numbers when they get large?

A If you have not experienced this phenomenon, see Exercise 2.1.9. The problem has to do with the way integers are represented. You will learn the details in Chapter 5. In the meantime, you can be safe with the strategy of using the `int` type when you know the values to be small and the `long` type if you think the values might get to be ten digits or more.

Q It seems wrong that Java should just let `int`s overflow and give bad values. Is there something we can do about that?

A Yes, we will revisit this issue in Section 2.2 and in Chapter 4. The short answer for now is that lack of such checking is one reason such types are called *primitive* types. But a little knowledge can go a long way in avoiding such problems. For example, it is fine to use the `int` type for small numbers, but when values run into the billions, you need to use `long`.

Q Are there some other operations besides the basic arithmetic operations defined for `int`s?

A There are a few defined in `Math`, such as `max` and `min`, but most functions in the library are defined for `double`s. You can sometimes use these by converting to `double` and back again, but it is often simpler to define you own function, as we will see in several examples in Section 2.4.

Q It is annoying to see all those digits when printing a `float` or a `double`. Can we get `System.out.println` to print out just two or three digits after the decimal point?

A That sort of task involves a closer look at the method used to convert from `double` to `String`. We will revisit this issue in Chapter 3; for now, we will live with the extra digits (which is not all bad, since doing so helps us to get used to the different primitive types of numbers).

Q How can you know for sure that `(a && b)` and `!(!a || !b)` are equivalent?

A Build the truth tables. You can use the definitions to evaluate the expressions for each of the four possibilities for the operands and see that the truth tables match.

Q Why does `10/3` give `3` and not `3.333333333`?

A Since both `10` and `3` are literal integers, Java sees not need for type conversion and uses integer division. You should write `10.0/3.0` if you mean the numbers to be `double` literals. If you write `10/3.0` or `10.0/3` Java does implicit conversion to get the same result.

Q What is the value of `Math.round(6.022E23)`?

A You should get in the habit of typing in a tiny Java program to answer such questions yourself (and trying to understand why your program produces the result that it does).

Q Can you compare a `double` to an `int`?

A Not without doing a type conversion, but you usually need not worry because Java usually does the requisite type conversion automatically. For example, if `x` is an `int` with the value `3` then the expression `(x < 3.1)` is `true`—Java converts `x` to `double` because `3.1` is a `double` literal and performs the comparison.

Q Can you use < and > to compare `String` variables?

A No. Those operators are defined only for primitive types.

Q How about `==` and `!=`?

A You can use `==` to test `String` operands for equality, but the result may not be what you expect, because of the distinction between a `String` and its value. For example, `"abc" == "ab" + x` is `false` when `x` is a `String` with value `"c"` because the two operands are different strings (albeit with the same value). This distinction for nonprimitive types is essential, as you will learn when we discuss methods for comparing `String` variables in Section X.

Q What is the result of division and remainder for negative integers?

A The quotient `a/b` is negative if and only if either `a` or `b` is negative; the remainder `a % b` is defined such that `(a/b)*b + a % b` is always equal to `a`. For example, `-14/3` and `14/-3` are both `-4`, but `-14 % 3` is `-2` and `14%-3` is `2`.

Q Are expressions like 1/0 and 1.0/0.0 legal in Java?

A No and yes. The first generates an arithmetic exception for division by 0; the second has the value `Infinity`.

Q Fifteen digits certainly seems enough to me. Do I really need to worry much about precision?

A Yes, because you are used to mathematics based on real numbers with infinite cardinality, while the computer always is dealing with approximations. For example, `(.1 + .1 == .2)` is `true` but `(.1 +.1 +.1 == .3)` is `false`! By printing out values of some simple expressions like these you can learn quite a bit about precision. For example, see Exercise 2.1.12.

Exercises

2.1.1 What does the following sequence of statements do?

```
t = x; x = y; y = t;
```

2.1.2 Write a program that uses `Math.sin` and `Math.cos` to check that $\sin\theta^2 + \cos\theta^2 = 1$ for any θ entered as a command-line argument.

2.1.3 Show that the expression `(!(a && b) && (a || b)) || ((a && b) || !(a || b))` is equivalent to `true`.

2.1.4 Simplify the following expression: `(!(a < b) && !(a > b))`

2.1.5 The *exclusive or* function for boolean variables is defined to be `true` if they are different, `false` if they are the same. Give a truth table for this function.

2.1.6 Explain how to use Program 2.1.3 to find the square root of a number.

2.1.7 What do each of the following print?

a `System.out.println(2 + "bc");`
b `System.out.println(2 + 3 + "bc");`
c `System.out.println((2+3) + "bc");`
d `System.out.println("bc" + (2+3));`
e `System.out.println("bc" + 2 + 3);`

2.1.8 What do each of the following print?

a `System.out.println ('b');`
b `System.out.println ('b' + 'c');`
c `System.out.println((char) ('a' + 4));`

2.1.9 Suppose that a variable `a` is declared as `int a = 2147483647`. What do each of the following print?

a `System.out.println(a);`
b `System.out.println(a+1);`
c `System.out.println(2-a);`
d `System.out.println(-2-a);`
e `System.out.println(2*a);`
f `System.out.println(4*a);`

Explain each outcome.

2.1.10 Suppose that a variable `a` is declared as `double a = 3.14159`. What do each of the following print?

a `System.out.println(a);`
b `System.out.println(a+1);`
c `System.out.println(8/(int) a);`
d `System.out.println((int) 8/a);`
e `System.out.println(8/a);`
f `System.out.println((int) (8/a));`

Explain each outcome.

2.1.11 Describe what happens if, in Program 2.1.3, you write `sqrt` instead of `Math.sqrt`.

2.1.12 What is the value of `(Math.sqrt(2)*Math.sqrt(2) == 2)`?

Creative Exercises

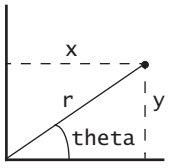
2.1.13 *Uniform random rumbers.* Write a program that uses `Math.random` to print ten uniform random values between 0 and 1 and then prints their average value.

2.1.14 *Wind chill.* Given the temperature T (in Fahrenheit) and the wind speed V (in miles per hour), the National Weather Service defines the effective temperature (the wind chill) to be:

$$W = 35.74 + 0.6125\,T + (0.475\,T - 35.75)\,V^{0.16}$$

Write a program that takes two integer command-line arguments T and V and prints out the wind chill. Hint: use `Math.pow(a, b)` to compute a^b.

2.1.15 *Loan payments.* Write a program that calculates the monthly payments you would have to make over a given number of years to pay off a loan at a given interest rate. Your program should take three numbers as command line parameters: the number of years, the principal, and the interest rate.



Polar coordinates

2.1.16 *Polar coordinates.* Write a program that converts from Cartesian to polar coordinates. Your program should take two real numbers x and y that are between 0 and 1 on the command line and print the polar coordinates r and θ. Use the Java methods `Math.sqrt` and `Math.asin`. Assume that x is not zero.

2.1.17 *Day of the week.* Write a program to read in a date and tell you what day of the week that date falls on. Your program should take three command line parameters, M (month), D (day), and Y (year). For M, use

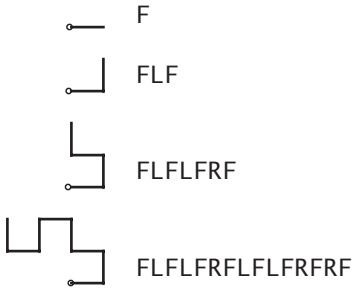
1 for January, 2 for February, and so forth. For output, print 0 for Sunday, 1 for Monday, 2 for Tuesday, and so forth. Use the following formulas, for the Gregorian calendar:

$$\begin{aligned} y &= Y - (14 - M)/12 \\ x &= y + y/4 - y/100 + y/400 \\ m &= M + 12 \times ((14 - M)/12) - 2 \\ d &= (D + x + (31m)/12) \,\%\, 7 \end{aligned}$$

Example: On what day of the week was August 2, 1953?

$$\begin{aligned} y &= 1953 - 0 = 1953 \\ x &= 1953 + 1953/4 - 1953/100 + 1953/400 = 2426 \\ m &= 8 + 12 \times 0 - 2 = 6 \\ d &= (2 + 2426 + (31 \times 6)/12) \,\%\, 7 = 0 \end{aligned}$$

Answer: Sunday.



Dragon curves of order 0, 1, 2, and 3

2.1.18 *Dragon curves.* Write a program to print the instructions for drawing the *dragon curves* of order 0 through 5. The instructions are strings of `F`, `L`, and `R` characters, where `F` means "draw a line 1 unit straight ahead," `L` means "turn left," and `R` means "turn right." A dragon curve of order n is formed when you fold a strip of paper in half n times, then unfold to right angles. The key to solving this problem is to note that a curve of order n is a curve of order $n - 1$ followed by an `L` followed by a curve of order $n - 1$ traversed in reverse order, then to figure out a similar description for a curve in reverse order.

2.2 Conditionals and Loops

We use the term *flow of control* to refer to the sequence of statements that are actually executed in a program. All of the programs that we have examined to this point have a simple flow of control: all of the statements in the program are executed, in the order given. In this section, we introduce statements that allow us to change the flow of control, using logic about the values of program varibles. This feature is an essential component of virtually every program we will write from this point forward.

Specifically, we consider Java statements that implement *conditionals*, where other statements (or groups of statements) may or may not be executed depending on certain conditions; and *loops*, where other statements (or groups of statements) may be executed multiple times, again depending on certain conditions. As you will see in numerous examples in this section, conditionals and loops truly harness the power of the computer and will equip you to write programs to accomplish a broad variety of tasks that you could not contemplate attempting without a computer.

If statements. Most computations require different actions for different inputs. To express these differences in Java, we use the `if` statement:

```
if (<boolean expression>) <statement>
```

This description introduces a formal notation that we will use from now on to specify the format of Java constructs. The notation is not difficult to understand; indeed, it is nearly self-explanatory. We put within angle brackets (<>) a construct that we have already defined, to indicate that we can use any instance of that construct where specified. In this case, the notation says that an `if` statement is the keyword `if`, followed by a boolean expression enclosed in parentheses, followed by a statement. The meaning of an `if` statement is also nearly self-explanatory: the statement after the boolean expression is to be executed if and only if the boolean expression is `true`.

We use <*statement*> to mean either a statement like an assignment statement or a method call (including a terminating semicolon) or a *block* of statements (a sequence of statements, each terminated by a semicolon,

Program 2.2.1 Simulating a coin flip

```
public class Flip
  {
    public static void main(String[] args)
      {
        if (Math.random() > 0.5)
             System.out.println("Heads");
        else System.out.println("Tails");
      }
  }
```

This program uses `Math.random` *to simulate a coin flip. Each time you run it, it prints either heads or tails. A sequence of flips will have many of the same properties as a sequence that you would get by flipping a coin, but it is not a random sequence.*

```
% javac Flip.java
% java Flip
Heads
% java Flip
Tails
% java Flip
Tails
```

enclosed in curly braces). We have already been using blocks in our programs: the body of `main` is a block.

As a simple example of the need for conditional statements, consider Program 2.1.4. You rarely will print out the value of a `boolean` as in that code; instead, you can use an `if` statement to produce more friendly output: If you replace the `System.out.println` statement in Program 2.1.4 by the three statements

```
System.out.print(year + " is ");
if (!isLeapYear) System.out.print("not ");
System.out.println("a leap year");
```

then the output is more informative. If you make this change and run the program with `java LeapYear 2003`, it will print `2003 is not a leap year`.

You can also add an `else` clause to an `if` statement, to express the concept of executing one statement or another, depending on whether the conditional is `true` or `false`:

`if (`*<boolean expression>*`)` *<statement T>* `else` *<statement F>*

Class `Flip` in Program 2.2.1 is an example of the use of the `if-else` statement, for the simple task of simulating a coin flip. To digress slightly, we note that this simple program introduces an interesting philosophical issue that is worth contemplating: Can we get a computer to produce random values? We will return to this question on several occasions, because having numbers with the properties of random numbers is critical for many practical applications.

The following table contains some more examples of the use of `if` and `if-else` statements. These examples are typical of simple calculations that you might need in programs that you write. They include examples of simple functions and checks for possible error conditions in input data. Since the semantics (meaning) of statements like these is similar to their meanings as natural-language phrases, you will quickly grow used to using them.

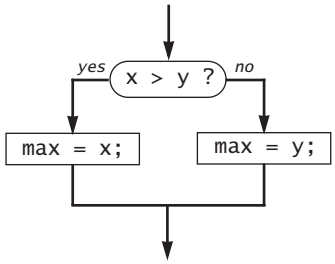
absolute value	`if (x < 0) x = -x;`
maximum of x and y	`if (x > y) max = x;` `else max = y;`

income tax rate	`if (income < 47450) rate = .22;`
	`else if (income < 114650) rate = .25;`
	`else if (income < 174700) rate = .28;`
	`else if (income < 311950) rate = .33;`
	`else rate = .35;`

error check before arithmetic operation	`if (d == 0)`
	`  System.out.println("Division by zero");`
	`else`
	`  System.out.println("Quotient is" + n/d);`

error check for quadratic formula	`d = b*b - 4.0*c;`
	`if (d < 0.0)`
	`  System.out.println("No real roots");`
	`else`
	`  {`
	`  d = Math.sqrt(d);`
	`  System.out.println((-b + d)/2.0);`
	`  System.out.println((-b - d)/2.0);`
	`  }`

The `if` and `if-else` statements have the same status as assignment statements or any other statements in Java. That is, we can use them whenever a statement is called for. However, when we use an `if` statement as `<statement T>` or `<statement F>` within another `if` statement (as in the income-tax-rate example in the table), we have to be careful to resolve ambiguities when doing so. This issue and more examples are addressed in the Q&A and exercises at the end of this section.



Flow of control for the statement

```
if (x > y) max = x;
else       max = y;
```

One way to understand control flow is to visualize it with a diagram called a flowchart, like the one at left. Paths through the flowchart correspond to flow-of-control paths in the program. In the early days of computing, when programmers used low-level languages and difficult-to-understand flows of control, flowcharts were an essential part of programming. With modern languages, we use flowcharts just to understand basic building blocks like the if-else statement.

While loops. Many computations are inherently repetitive. The basic Java construct for handling such computations has the following format:

```
while (<boolean expression>) <statement>
```

The `while` statement has the same form as the `if` statement (the only difference being the use of the keyword `while` instead of `if`), but the meaning is quite different. It is an instruction to the computer to behave as follows: if the expression is `false`, do nothing; if the expression is `true`, execute the statement (just as with `if`) but then perform the same operation and continue until the expression is not `true` any more. In other words, the `while` statement is equivalent to a sequence of identical `if` statements:

```
if (<boolean expression>) <statement>
if (<boolean expression>) <statement>
if (<boolean expression>) <statement>
 ...
```

Program 2.2.2 Your first while loop

```
public class TenHellos
  {
    public static void main(String[] args)
      {
        System.out.println("1st Hello");
        System.out.println("2nd Hello");
        System.out.println("3rd Hello");
        int i = 4;
        while (i <= 10)
          {
            System.out.println(i + "th Hello");
            i = i+1;
          }
  }
```

This program uses a `while` *loop for the simple repetitive task of printing the ouput shown at right. After the third line, the lines to be printed differ only in the value of the index counting the line printed, so we define a variable to contain that index and increment it each time through the loop to get the job done.*

```
% javac TenHellos.java
% java TenHellos
1st Hello
2nd Hello
3rd Hello
4th Hello
5th Hello
6th Hello
7th Hello
8th Hello
9th Hello
10th Hello
```

At some point, the statement must change something (such as the value of some variable in the expression) to make the expression `false`, and then the sequence is broken.

There is a profound difference between programs with `while` statements and programs without them, because `while` statements allow us to specify a potentially unlimited number of different possible sequences of statements to be executed in a program. In particular, the `while` statement allows us to specify lengthy computations in short programs. This ability opens the door to writing programs for all manner of tasks. But there is also

Program 2.2.3 Computing powers of two

```
public class PowersOfTwo
  {
    public static void main(String[] args)
      {
        int i = 0;
        int N = 1;
        while (i <= 30)
          {
            System.out.println(i + " " + N);
            i = i + 1;
            N = 2 * N;
          }
      }
  }
```

This program prints a table of the powers of two in the Java `int` *data type. Each time through the loop, the value of* `i` *is incremented by* `1` *and the value of* `N` *is doubled. To save space, we show only the first four and the last two lines of the table at right; the program prints a 31-line table.*

```
% javac PowersOfTwo.java
% java PowersOfTwo
0 1
1 2
2 4
3 8
...
29 536870912
30 1073741824
```

a price to pay: as your programs become more sophisticated, they become more difficult to understand.

For a simple example that uses a `while` statement, consider class `TenHellos` in Program 2.2.2, which performs a simple task that could be accomplished with a sequence of nearly identical `System.out.println` statements. According to the template just given, the `while` loop is equivalent to a sequence of copies of the statement

```
if (i <= 10) { System.out.println(i + "th Hello"); i = i+1; }
```

In this case, the statement is executed seven times and the sequence broken when the value of `i` becomes `11`. Using the `while` loop is barely worthwhile

for this specific simple task, but you will very soon be addressing tasks where you will need to use `while` loops to specify that statements be repeated far too many times to contemplate doing so any other way.

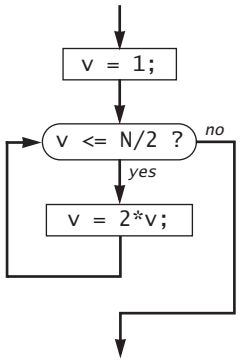
Class `PowersOfTwo` in Program 2.2.3 uses a `while` loop to print out a table of the powers of 2. This code is the prototype for many useful computations: by varying the computations that change the accumulated value and the way that the loop control variable is incremented, we can print out tables of a variety of functions (see Exercise 2.2.10 and Exercise 2.2.11).

It is worthwhile to examine the behavior of Program 2.2.3 carefully, to learn how to think about the behavior of programs with loops. A tried-and-true method for checking that a program is behaving as we expect is known as *tracing* the program. For example, the following table is a trace of the operation of Program 2.2.3. It shows the value of each of the variables before each iteration of the loop (and, for emphasis, the value of the conditional expression that controls the loop).

i	*N*	*i <= 30*	*i*	*N*	*i <= 30*	*i*	*N*	*i <= 30*
0	1	true						
1	2	true	11	2048	true	21	2097152	true
2	4	true	12	4096	true	22	4194304	true
3	8	true	13	8192	true	23	8388608	true
4	16	true	14	16384	true	24	16777216	true
5	32	true	15	32768	true	25	33554432	true
6	64	true	16	65536	true	26	67108864	true
7	128	true	17	131072	true	27	134217728	true
8	256	true	18	262144	true	28	268435456	true
9	512	true	19	524288	true	29	536870912	true
10	1024	true	20	1048576	true	30	1073741824	true
						31	0	false

At first, tracing can seem tedious, but it is worthwhile because it clearly exposes what a program is doing. This program is nearly a self-tracing program, because it prints the values of its variables each time through the loop.[1]

How did we know to stop the loop at `30`? From Appendix X, we know that the largest integer in Java's `int` data type is $2^{31} - 1$. This value is also available to our programs as `Integer.MAX_VALUE`, so we could change the test to `(N < Integer.MAX_VALUE/2)` to be sure that we stop before printing an incorrect value.



Flow of control for the sequence

```
v = 1;
while (v <= N/2) v = 2*v;
```

As a more complicated example, suppose that we want to compute the largest power of two not larger than a given integer `N`. For example, if `N` is `13` we want the result `8`, if `N` is `1000`, we want the result `512`, if `N` is `64` we want the result `64`, and so forth. This computation is simple to perform with a `while` loop, as follows:

```
v = 1;
while (v <= N/2) v = 2*v;
```

It takes some thought to convince oneself that this simple piece of code produces the desired result. You can do so in four steps: first, `v` is always a power of `2`; second, `v` is never greater than `N`; third, `v` increases each time through the loop, so the loop must eventually terminate; and fourth, after the loop terminates, `2*v` *is* greater than `N`. Reasoning of this sort is often important in understanding how `while` loops work. Even though many of the loops you will write are much simpler than this one, you should be sure that you understand each loop that you write.

When you have a program that uses loops but does not print out intermediate values, you will find yourself debugging it by adding statements that do so, to produce a trace. In other words, when working with loops, it is best to think about how the values of the variables change each time through the loop. You may wish to reinforce this way of thinking in your first few pro-

1. There are many situations in computer science where it is useful to be familiar with powers of two. You should know at least the first 10 values in this table and you should note that 2^{10} is about 1 thousand, 2^{20} is about 1 million, and 2^{30} is about 1 billion.)

grams that use loops by printing out the values, as does Program 2.2.3. As you gain more confidence, you will use traces that print out the values of variables only at critical junctures during the execution of the program.

For loops. As you will see, the `while` loop suffices to allow us to write programs for all manner of applications. Before considering more examples, we will look at alternate Java notations and constructs that allow us more flexibility when writing programs with loops. These alternate ways of specifying loops do not provide anything fundamentally different from the basic `while` loop, but they allow us to write more compact and natural programs than if we used only `while` statements.

For notation. Many loops follow the same basic scheme: initialize an index variable to some value and then use a `while` loop to test an exit condition involving the index variable, using the last statement in the `while` loop to modify the index variable. Java's `for` notation is a direct way to express such loops. The statement

```
for (<init stmnt>; <boolean expression>; <incr stmnt>)
  { <body statements> }
```

is, with only a few exceptions, *equivalent* to the sequence

```
<init stmnt>;
while (<boolean expression>)
  { <body statements> <incr stmnt> }
```

There is usually no difference between a `for` loop and its corresponding `while` loop—your Java compiler might even produce identical object code for the two loops. When there is only one `<body statement>`, the braces are not needed in the `for` statement (and a semicolon is needed to separate it from the `<incr stmnt>` in the `while` statement).

The main purpose of the `for` loop is to support a typical programming idiom: perform a computation for a sequence of values of a variable. For example, the following two lines of code are equivalent to the final six lines of code in `TenHellos` (Program 2.2.2):

```
for (i = 4; i <= 10; i = i+1)
  System.out.println(i + "th Hello");
```

Typically, we work with an even more compact version of this code, using the shorthand notation discussed next.

Idioms for single-operand operations. Modifying the value of a variable is something that we do so often in programming that Java provides a variety of different shorthand notations for the purpose. For example, the following four statements all increment the value of `i` by `1` in Java:

```
i = i + 1;     i++;      ++i;     i += 1;
```

You can also say `i--` or `--i` or `i -= 1` for `i = i-1`. Most programmers use `i++` or `i--` in `for` loops, though any of the others would do. The `++` and `--` constructs are normally used for integers, but the "operator-equals" construct are useful operations for any arithmetic operator in any primitive numeric type. For example, you can say `N *= 2` or `N += N` instead of `N = 2*N`. All of these idioms are for notational convenience, nothing more.

Statement sequences. The statements in the for loop header can be *sequences* of statements, separated by commas. This notation allows us to initialize and modify other variables besides the loop index in the header. For example, the following two lines of code could replace the eight-line body of the `main` method in `PowersOfTwo` (Program 2.2.3):

```
for (i = 0, N = 1; i <= 30; i++, N *= 2)
  System.out.println(i + " " + N);
```

Null statements. Not all parts of a `for` loop need to be filled in with code. For example, the following three lines of code are all equivalent (they all set `v` to the largest power of 2 not larger than `N`):

```
v = 1; while (v <= N/2) v *= 2;
for (v = 1; v <= N/2; ) v *= 2;
for (v = 1; v <= N/2; v *= 2) ;
```

The first is the `while` statement that we considered as an example earlier in this section; the second has a null increment statement; the third has a null loop body.

Choosing among different formulations of the same computation like these is a matter of each individual programmer's taste, as when a writer picks from among synonyms or chooses between using active and passive

voice when composing a sentence. Some programmers will passionately defend one usage or another, just as some writers will passionately defend one usage or another. You won't find a good hard-and-fast rule on how to compose a program that expresses a computation any more than you will find good hard-and-fast rules on how to compose a paragraph that expresses a thought. Your goal should be to find a style that suits you, gets the computation done, and can be appreciated by others.

This combination of shortcuts came into widespread use with the C programming language in the 1970s and have become standard. They have survived the test of time because they lead to compact, elegant and easily understood programs. When you master the art of creating such programs, you will be able to transfer that skill to programming in numerous modern languages, not just Java.

Applications. The ability to program with loops immediately opens up the world of computation to us. To emphasize this fact, we next consider a variety of examples. These examples all involve working with the primitive types of data (and `Strings`) that we considered in Section 2.1, but rest assured that the same mechanisms serve us well for any computational application.

These sample programs are carefully crafted—by studying and appreciating them, you will be in good position to write your own programs containing loops, as requested in many of the exercises at the end of this section.

Program 2.2.4 Ruler subdivisions

```
public class RulerN
  {
    public static void main(String[] args)
      {
        int N = Integer.parseInt(args[0]);
        String ruler = " ";
        for (int i = 1; i <= N; i++)
          ruler = ruler + i + ruler;
        System.out.println(ruler);
       }
  }
```

This program uses a for loop to simplify the iterative computation in Program 2.1.1. *Note that Java automatically converts* `i` *to a string each time through the loop.*

```
% javac RulerN.java
% java RulerN 2
 1 2 1
% java RulerN 4
 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

Ruler subdivisions. Program 2.2.4 uses a `for` loop to compute the string of ruler subdivision lengths that we considered in Program 2.1.1, but taking the number of subdivisions from the command line. This program is an illustration of one of the essential charactersitics of loops—the program could hardly be simpler, but it can produce a huge amount of output. You might wish to trace this program as a warmup to understanding the ones that follow. Its simplicity is deceptive.

The rest of the examples that we consider involve computing with numbers, which has the advantage that the problems that arise are natural if not familiar ones. Several of our examples are tied to classical problems faced by mathematicians and scientists as they have learned and applied properties of numbers through the centuries. While computers have existed for only 50 years or so, many of the fundamental computational methods that we use are based on a rich mathematical tradition tracing back to antiquity.

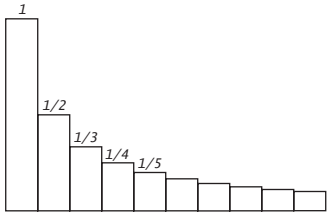
Program 2.2.5 Computing the Harmonic numbers

```
public class Harmonic
  {
    public static void main(String[] args)
      {
        int N = Integer.parseInt(args[0]);
        double sum = 0.0;
        for (int i = 1; i <= N; i++)
          sum += 1.0/i;
        System.out.println(sum);
      }
  }
```

This program computes the value of the Nth Harmonic number. The value is known from mathematical analysis to be about ln(N) + .57721... for large N. As a check, note that ln(10000) = 9.21034...

```
% javac Harmonic.java
% java Harmonic 2
1.5
% java Harmonic 10
2.9289682539682538
% java Harmonic 10000
9.787606036044348
```

Finite sum. The computational paradigm used by Program 2.2.3 is one that you will use frequently. It uses two variables—one as an index that controls a loop and the other to accumulate a computational result.



Class `Harmonic` in Program 2.2.5 uses the same paradigm to evaluate the sum

$$H_N = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{N}$$

These numbers, which are known as the *Harmonic numbers*, arise frequently in mathematical analysis. Harmonic numbers are the discrete analog of the logarithm: they approximate the area under the curve $y = 1/x$. This approximation is a special case of *quadrature*, or numerical integration, which we will consider in Chapter 9.

You can easily use Program 2.2.5 as a model for computing the values of other sums (see Exercise 2.2.16).

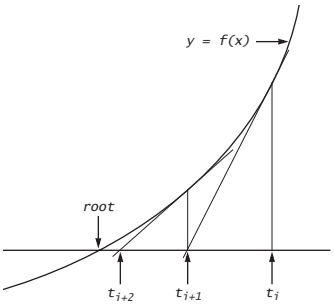
Program 2.2.6 Newton's method for computing the square root

```
public class Sqrt
  {
    public static void main(String[] args)
      {
        double c = Double.parseDouble(args[0]);
        double t = c;
        while (t - c/t > .000000000000001)
          t = (c/t + t)/2.0;
        System.out.println(t);
      }
  }
```

This program computes the square root of its command-line argument, using Newton's method (see text). It stops when it has an answer accurate to within 15 decimal places.

```
% javac Sqrt.java
% java Sqrt 2
1.414213562373095
java Sqrt 1048575
1023.9995117186336
```

Computing the square root. How are functions in Java's `Math` library such as `Math.sqrt` implemented? Class `Sqrt` in Program 2.2.6 illustrates one technique. To compute the square root function, it uses an iterative computation that is derived from a general computational technique that was developed in the 17th century by Isaac Newton and Joseph Raphson and is widely known as *Newton's method*.



Newton's method

Under generous conditions on a given function, Newton's method is an effective way to find roots (values of x for which the function is 0): Start with an initial estimate t_0. Given an estimate t_i, compute a new estimate by drawing a line tangent to the curve $y = f(x)$ at the point $(t_i, f(t_i))$ and set t_{i+1} to the x-coordinate of the point where that line hits the x axis. As we iterate this process, we get closer to the root.

Computing the square root of a positive number c is equivalent to finding the positive root of the function $f(x) = x^2 - c$. It is a simple exercise in analytic geometry (see Exercise 2.2.17) to show that Newton's method for this problem amounts to the process implemented in Program 2.2.6: Start with an estimate t. If t is equal to c/t, then t is equal to the square root of c, so the computation is complete. If not, refine the estimate by replacing t with the average of t and c/t :

N	*t*	*c/t*
1	2.0000000000000000	1.0
2	1.5000000000000000	1.3333333333333333
3	1.4166666666666665	1.4117647058823530
4	1.4142156862745097	1.4142114384748700
5	1.4142135623746899	1.4142135623715002
6	1.4142135623730950	1.4142135623730951

With Newton's method, we get the value of $\sqrt{2}$ accurate to 15 places in just 5 iterations of the loop.

Newton's method is important in scientific computing because the same iterative approach is effective for finding roots of a broad class of functions, including many for which analytic solutions are not known (so the Java `Math` library would be no help). We cover Newton's method more thoroughly and consider applications in Section 9.1.

With computers and calculators, we take for granted that we can find whatever values we need of mathematical functions; before computers, scientists and engineers had to use tables or computed values by hand. Not surprisingly, computational techniques that were developed to enable such calculations needed to be very efficient, so it is not suprising that many of those same techniques are effective when we use computers. Newton's method is a classic example of this phenomenon. Another useful approach for evaluating mathematical functions is to use *Taylor series* expansions (see Exercise 2.2.36 and Exercise 2.2.37).

Program 2.2.7 Converting to binary representation

```
public class Binary
  {
    public static void main(String[] args)
      {
        int v, N = Integer.parseInt(args[0]);
        for (v = 1; v <= N/2; v *= 2) ;
        for ( ; v > 0; v /= 2)
          if (N < v) { System.out.print(0); }
          else       { System.out.print(1); N -= v; }
        System.out.println();
      }
  }
```

This program prints the binary representation of the command-line argument, by casting out powers of 2 in decreasing order (see text). Note that the first `for` *loop has a null loop body and the second* `for` *loop has a null initialization statement.*

```
% javac Binary.java
% java Binary 5
101
% java Binary 106
11001010
% java Binary 100000000
101111101011110000100000000
```

Number conversion. Class `Binary` in Program 2.2.7 prints the binary (base 2) representation of the decimal number typed as the command-line argument. It is based on decomposing the number into a sum of powers of two. For example, the binary representation of 106 is 1101010, which is the same as saying that 106 = 64+32+8+2. In binary, this equation is more transparent:

```
   1000000
+   100000
+     1000
+       10
---------
= 1101010
```

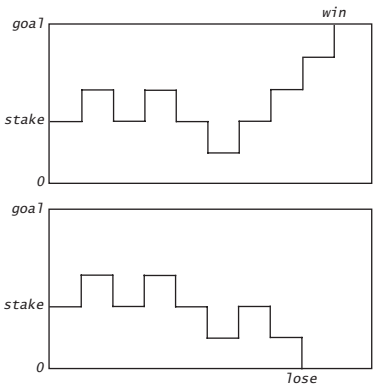
To compute the binary representation of `N`, we consider the powers of 2 less than or equal to `N` in decreasing order to determine which belong in the binary decomposition (and therefore correspond to a 1 bit in the binary representation). A trace (of the second `for` loop in the program for the case

when N is 106) will be instrumental in helping you to understand how the program does its job.

N	*binary representation*	v	*binary representation*	N < v	*output*
106	1101010	64	1000000	false	1
42	101010	32	100000	false	1
10	01010	16	10000	true	0
10	1010	8	1000	false	1
2	010	4	100	true	0
2	10	2	10	false	1
0	0	1	1	true	0
0		0			

Read from top to bottom inthe rightmost column, the output is 1101010, the binary representation of 106.

Converting data from one representation to another is a frequent theme in writing computer programs. Thinking about conversion emphasizes the distinction between an abstraction (an integer like the number of hours in a day) and a representation of that abstraction (24 or 11000). The irony in this case is that the computer's representation of an integer is actually based on its binary representation, as we shall see in Chapter 5.



Gambler simulation sequences

Simulation. Our next example is different in character from the ones we have been considering, but it is representative of a common situation where we use computers to simulate what might happen in the real world, so that we can make informed decisions in all kinds of complicated situations. The specific problem that we consider now is from a thoroughly-studied class of problems known as *gambler's ruin* problems. Suppose that a gambler makes

Program 2.2.8 Gambler simulation

```
public class Gambler
  {
    public static void main(String[] args)
      {
        int stake = Integer.parseInt(args[0]);
        int goal  = Integer.parseInt(args[1]);
        int N     = Integer.parseInt(args[2]);
        int bets = 0, wins = 0;
        for (int i = 0; i < N; i++)
          { int t;
            for (t = stake; t > 0 && t < goal; bets++)
              if (Math.random() > 0.5)
                   t++;
              else t--;
            if (t == goal) wins++;
          }
        System.out.println(100*wins/N + "% wins");
        System.out.println("Avg # bets: " + bets/N);
      }
  }
```

The inner for loop in this program simulates a gambler with `$stake` *who makes a series of* `$1` *bets, continuing until going broke or reaching* `$goal`.

```
% javac Gambler.java
% java Gambler 10 20 1000
50% wins
Avg # bets: 100
% java Gambler 50 250 100
19% wins
Avg # bets: 11050
```

a series of fair $1 bets, starting with some given initial stake. The gambler always goes broke, eventually, but when we set other limits on the game, various questions arise. For example, suppose that the gambler decides ahead of time that she will walk away after reaching a certain goal. What are the chances that she will win, and how many bets might she have to make before winning or losing the game?

Class `Gambler` in Program 2.2.8 is a simulation that can help answer these questions. It does a sequence of trials, using `Math.random` to simulate the sequence of bets, continuing until the gambler is broke or the goal

reached, and keeping track of the number of wins and the number of bets. After running the experiment for the specified number of trials, it averages and prints out the results. You might wish to run this program for various values of the command-line arguments, not necessarily just to plan your next trip to the casino, but to help you think about the following questions: Is the simulation an accurate reflection of what would happen in real life? How many trials are needed to get an accurate answer? What are the computational limits on performing such a simulation? Simulations are widely used in applications in economics, science, and engineeering, and questions of this sort are important in any simulation.

In the case of Program 2.2.8, we are merely verifying classical results from probablity theory, which say the the probability of success is the ratio of the stake to the goal and that the expected number of bets is the product of the stake and the desired gain (the difference between the goal and the stake). For example, if you want to go to Monte Carlo to try to turn $500 into $2500, you have a reasonable (20%) chance of success, but you should expect to make a million $1 bets!

Simulation and analysis go hand-in-hand, each validating the other. In practice, the value of simulation is that it can suggest answers to questions that might be too difficult to resolve with analysis. For example, suppose that our gambler, recognizing that she will never have the time to make a million bets, decides ahead of time to set an upper limit on the number of bets. How much money can she expect to take home in that case? You can address this question with an easy change to Program 2.2.8 (see Exercise 2.2.25), but addressing it with analysis is likely to be beyond your reach.

Program 2.2.9 Factoring integers

```
public class Factors
  {
    public static void main(String[] args)
      {
        long N = Long.parseLong(args[0]);
        for (long i = 2; i <= N/i; i++)
          while (N % i == 0)
            { N /= i; System.out.print(i + " "); }
        if (N > 1) System.out.println(N);
      }
  }
```

This program prints the prime factorization of any positive integer in Java's `long` *data type. The code is simple, but it takes some thought to convince oneself that it is correct (see text).*

```
% javac Factors.java
% java Factors 3757208
2 2 2 7 13 13 397
% java Factors 287994837222311
17 1739347 9739789
```

Factoring. Class `Factors` in Program 2.2.9 computes the prime factorization of any given positive integer. In contrast to many of the other programs that we have seen (which we could do in a few minutes with a calculator or even a pencil and paper), this computation would not be feasible without a computer. How would you go about trying to find the factors of a number like `287994837222311`? Even with a calculator, you might find the factor `17` quickly, but it would take you quite a while to find `1739347`.

While the program is compact and straighforward, it takes some thought to be convinced even that this program produces the desired result for any given integer, so we turn to that task next.

As usual, we begin with a trace. The following table shows the values of the variables just before each iteration of the outer `for` loop when Program 2.2.9 is invoked with the command `java Factors 3757208`. When `i` is `2`, the inner `for` loop is iterated three times to remove the three factors of `2`; when `i` is `3`, `4`, `5`, and `6`, the inner `for` loop is iterated zero times since none of those numbers divide `46951`; and so forth.

i	*N*	*output*
2	3757208	2 2
3	469651	
4	469651	
5	469651	
6	469651	
7	469651	7
8	67093	

i	*N*	*output*
9	67093	
10	67093	
11	67093	
12	67093	
13	67093	13 13
14	397	
15	397	

i	*N*	*output*
16	397	
17	397	
18	397	
19	397	
20	397	
		397

This trace clearly exposes the basic operation of the program. To convince ourselves that it is correct, we reason at a higher level of abstraction about what we expect each of the loops to do. The `while` loop clearly prints and removes from `N` all factors of `i`—the key to understanding the program is to see that the following fact holds at the beginning of each iteration of the `for` loop: `N` *has no nontrivial factors less than* `i`. Thus, if `i` is not prime, it will not divide `N`; if `i` is prime, the `while` loop will do its job. Moreover, once we know that `N` has no factors less than or equal to `i`, we also know that it has no factors greater than `N/i`, so we need look no further when `i` is greater than `N/i`.

In a more naive implementation, we might simply have used the condition `(i < N)` to terminate the `for` loop. Even given the blinding speed of modern computers, such a decision would have a dramatic effect on the size of the numbers that we could factor. Exercise 2.2.27 encourages you to experiment with the program to learn the effectiveness of this simple change. On a computer that can do billions of operations per second we could do numbers on the order of 10^9 in a few seconds; with the `(i <= N/i)` test we can do numbers on the order of 10^{18} in a comparable amount of time. Loops give us the ability to solve difficult problems, but they also give us the ability to construct simple programs that run slowly, so we must always be congizant of performance.

In modern applications such as cryptography (see Chapter 10), there are actually situations where we wish to factor truly huge numbers (with, say,

hundreds or thousands of digits). Such a computation is difficult enough even *with* the use of a computer—we will return to this problem in Chapter 8.

Other conditional and loop constructs. To be complete, we consider here three more Java-language constructs that relate to loops. You need not be thinking about using these constructs for every program that you write, because they used much less frequently than the `if`, `while` and `for` statements. But it is worthwhile to be aware of them, because it is often the case that using one of them can be helpful when you get stuck or find yourself with overly complicated code trying to compose a conditional or a loop.

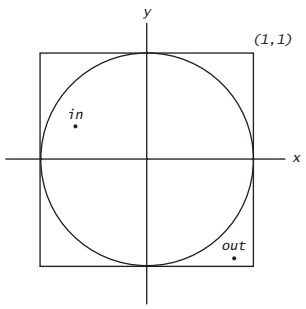
Do-while notation. Another way to write a loop is to use the statement

```
do <statement> while (<condition>);
```

The meaning of this statement is the same as

```
while (<condition) <statement>
```

except that the first test of the condition is omitted. If the condition initially holds, then there is no difference.



For example, the following code sets x and y such that (x, y) is randomly distributed in the unit disk.

```
do
  {
    x = 2.0*Math.random() - 1.0;
    y = 2.0*Math.random() - 1.0;
  }
while (Math.sqrt(x*x + y*y) > 1.0);
```

With `Math.Random`, we get points that are randomly distributed in the unit *square*—we just generate points until finding one in the unit disk. We always want to generate at least one point, so a `do-while` loop is called for. Since the area of the disk is π and the area of the square is 4, the expected number of times the loop is iterated is just $4/\pi$ (about 1.27). There are a few other similar examples that we will point out when we encounter them later in the book, but you do not need to worry about `do-while` loops now. Most programmers rarely use them.

Break statement. In some situations, we want to immediately exit a loop, without letting it run to completion. Java provides the `break` statement for this purpose. For example, the following code is an effective way to test whether a given integer `N` is prime:

```
int i;
for (i = 2; i <= N/i; i++)
  if (N % i == 0) break;
if (i > N/i) System.out.println("N is prime");
```

There are two different ways to leave this loop: either the `break` statement was executed (because `i` divides `N`, so `N` is not prime) or the `for` loop condition was not satisfied (so no `i` with `i <= N/i` was found that divides `N`, which implies that `N` is prime). Note that we have to declare `i` outside the `for` loop instead of in the initialization statement.

Continue statement. Java also provides a way to skip to the next iteration of a loop: the `continue` statement. When a `continue` is executed within a loop body, the flow of control immediately transfers to the next iteration of the loop. Usually, it is easy to achieve the same effect with an `if` statement inside the loop. The `continue` statement provides the exception to the rule that `while` and `for` statements are equivalent.

Switch statement. The `if` and `if-else` statements allow one or two alternatives in directing the flow of control. Sometimes, a computation naturally suggests more than two alternatives. We can use a sequence or a chain of `if-else` statements in such situations, but the Java `switch` statement provides a direct solution. We omit a formal description and move right to a typical example: Rather than printing an `int` variable `day` in a program that works with days of the weeks (such as a solution to Exercise 2.1.17) it would be preferable to use a `switch` statement, as follows:

```
switch (day)
  {
    case 0: System.out.println("Sunday");    break;
    case 1: System.out.println("Monday");    break;
    case 2: System.out.println("Tuesday");   break;
    case 3: System.out.println("Wednesday"); break;
    case 4: System.out.println("Thursday");  break;
```

```
            case 5: System.out.println("Friday");   break;
            case 6: System.out.println("Saturday"); break;
        }
```

We leave details beyond this example to Appendix X. When you have a progam that seems to have a long and regular sequence of `if` statements, you might consider consulting Appendix X and using a `switch` statement. We will also consider another option in Section 2.3.

Generally, you do not need to worry about using the `do-while`, `break`, `continue`, or `switch` statements until you are comfortable using `if`, `while`, and `for`. Many programmers do not use them at all.

Infinite loops. Before you write programs that use loops, you need to think about the following issue: What if the condition that is supposed to end a `while` loop is never satisfied? One of two bad things could happen, both of which you need to learn to cope with.

First, suppose that such a loop calls `System.out.println`. For example, if the condition in `TenHellos` were `(i > 3)` instead of `(i <= 10)`, it would always be `true`. What happens? Nowadays, we use "print" as an abstraction to mean "display in a terminal window" and the result of attempting to display an unlimited number of lines in a terminal window is dependent on operating-system conventions. If your system is set up to have "print" mean "print characters on a piece of paper" you might run out of paper or have to unplug the printer. In a terminal window, you need a "stop printing" operation that has a similar effect. Before running programs with loops on your own, you should find out what you should do to "pull the plug" on an infinite loop of `System.out.println` calls (see the booksite for instructions on several widely-used systems) and then test out the strategy by making the change to `TenHellos` indicated above and trying to stop it.

Second, *nothing* might happen. If your program has an infinite loop that does not consume any resources, it will happily spin through the loop and you will see no results at all. When you find yourself in such a situation, you can inspect the loops to make sure that the loop exit condition always happens, but the problem may not be easy to indentify. One way to locate such a bug is to insert calls to `System.out.println` to produce a trace. If

these calls fall within an infinite loop, this strategy reduces the problem to the case discussed in the previous paragraph!

You might not know (or it might not matter) whether a loop is infinite or just very long. For example, if you invoke Program 2.2.8 with large arguments such as `java Gambler 100000 200000` you may not want to wait for the answer. You will learn to be aware of and to estimate the running time of our programs.

Why not have the Java compiler detect infinite loops and warn us about them? You might be surprised to know that it is *not possible* to do so, in general. This fact is one of the fundamental results of theoretical computer science, which we will address in Chapter 8.

Summary. For reference, the following table lists and characterizes the programs that we have considered in this section. They are representative of the kinds of tasks we can address with short programs comprised of `if`, `while` and `for` statements processing simple types of data. You will find several more examples in the exercises. These types of computations are an appropriate way to become familar with basic Java flow-of-control constructs such as the `if`, `while`, and `for` statements. The time that you spend now working with as many such programs as you can will certainly pay off for you in the future.

`TenHellos`	*your first loop*
`PowersOfTwo`	*compute and print a table of values*
`RulerN`	*short program produces long output string*
`Harmonic`	*compute finite sum*
`Newton`	*classic iterative algorithm*
`Binary`	*basic number conversion*
`Gambler`	*simulation experiment with nested* `for` *loops*
`Factor`	`while` *loop within a* `for` *loop*

To learn how to use conditionals and loops, you must practice writing and debugging programs with `if`, `while`, and `for` statements. The exercises at the end of this section provide many opportunities for you to begin this process. Some involve modifying and extending the programs in this section; others present new challenges.

For each exercise, you will write a Java program, then run and test it. All programmers know that it is unusual to have a program work as planned the first time it is run, so you will want to have an understanding of your program and an expectation of what it should do, step by step. At first, use explicit traces to check your understanding and expectation. As you gain experience, you will find yourself thinking in terms of what a trace might produce as you compose your loops. Ask yourself the following sorts of questions: What will be the values of the variables after the loop interates the first time? The second time? The final time? Is there any way this program could get stuck in an infinite loop?

Loops and conditionals are a giant step in our ability to compute: `if`, `while` and `for` statements take us from simple straight-line programs to arbitrarily complicated flow of control. In the next several chapters, we will take more giant steps: to allow us to process large amounts of input data and to allow us to define and process other types of data than simple numeric types. The if, while and for statements of this section will play an essential role in the programs that we consider as we take these steps forward.

Q&A

Q What is the difference between = and ==?

A That was our first question after Section 2.1, but we repeat it here to remind you to be sure not to use = when you mean == in a conditional expression. The expression `(x = y)` assigns the value of `y` to `x` and also is a conditional expression with value `true`, so `if (x = y) <statement>` does not test whether `x` is equal to `y` but rather unconditionally does the assignment and executes the statement. In lower-level programming languages, this difference can wreak havoc in a program and be difficult to detect, but Java's type safety usually will come to the rescue. For example, if

we made the mistake of typing `(t = goal)` instead of `(t == goal)` in Program 2.2.8, the compiler would find the bug for us:

```
javac Gambler.java
Gambler.java:41: incompatible types
found   : int
required: boolean
if (t = goal) wins++;
    ^
1 error
```

Q So using == instead of = is something I need to pay attention to when writing loops and conditionals. Is there something else in particular that I should watch out for?

A Another very common mistake is to forget the braces in a loop or conditional with a two-statement body. For example, consider this version of the code in `Gambler`

```
for (int i = 0; i < trials; i++)
  for (t = stake; t > 0 && t < goal; bets++)
    if (Math.random() > 0.5)
         t++;
    else t--;
    if (t == goal) wins++;
```

The code looks good, but is dsyfunctional because the second `if` is outside both `for` loops and gets executed just once.

Q Anything else?

A The third classic pitfall is ambiguity in nested `if` statements. If you write

```
if <expr1> if <expr2> <stmntA> else <stmntB>
```

you could mean either that `<stmntB>` is to be executed if `<expr1>` is `false`:

```
if <expr1> { if <expr2> <stmntA> } else <stmntB>
```

or that `<stmntB>` is to be executed if `<expr1>` is `true` and `<expr2>` is `false`:

```
if <expr1> { if <expr2> <stmntA> else <stmntB> }
```

When we write the statements on one line as above, the ambiguity is obvious; real code with indenting is susceptible to insidious bugs like the one described in the last question. The rule in Java is that `else` always refers to the most recent if that has no `else` (the second meaning above). To avoid this trap, it is good programming practice to use explicit braces in such situations.

Q Is there some program for which I *must* use a `for` loop but not a `while` loop, or vice versa?

A No.

Q What are the rules on where we declare the loop-control variables?

A Opinions differ. In older programming languages, it was required that all variables be declared at the beginning of a block, so many programmers are in this habit and there is a lot of code out there that follows this convention. But it makes a lot of sense to declare variables where they are first used, particularly in `for` loops, when it is normally the case that the variable is not needed outside the loop. But is also not uncommon to need to test (and therefore declare) the loop-control variable outside the loop, as in Program 2.2.8.

Q What is the difference between `++i` and `i++`?

A As statements, no difference. In expressions, both increment `i` but `++i` has the value after the increment and `i++` the value before the increment. It is safe for you not worry much about this distinction and just use `i++` for now. When we use `++i` in this book, we will call attention to it and say why.

Exercises

2.2.1 Write a program that reads in three integer parameters and prints `equal` if all three are equal, and `not equal` otherwise.

2.2.2 What is wrong with each of the following statements?

a
```
if (a > b) then c = 0;
```

b
```
if a > b { c = 0; }
```

c
```
if (a > b) c = 0;
```
d
```
if (a > b) c = 0 else b = 0;
```

2.2.3 Write a more general and more robust version of `Quadratic` (Program 2.1.3) that prints the roots of the polynomial $ax^2 + bx + c$ and also prints an appropriate message if the discriminant is negative, and behaves appropriately (avoiding division by zero) if a and/or b is zero.

2.2.4 Extend your solution to Exercise 2.1.16 to convert (x, y) to polar coordinates for any values of x and y.

2.2.5 Suppose that `i` and `j` are both of type `int`. What is the value of `j` after each of the following statements is executed?

a
```
for (i = 0, j = 0; i < 10; i++) j += i;
```
b
```
for (i = 0, j = 1; i < 10; i++) j += j;
```
c
```
for (j = 0; j < 10; j++) j += j;
```
d
```
for (i = 0, j = 0; i < 10; i++) j += j++;
```

2.2.6 Rewrite `TenHellos` to make a program `Hellos` that takes the number of lines to print as a command-line argument. You may assume that the argument is less than 1000. Hint: Use `i % 10` and `i % 100` to determine when to use `st`, `nd`, `rd`, or `th` for printing the `i`th `Hello`.

2.2.7 Write a program that, using one `for` loop and one `if` statement, prints the integers from 1000 to 2000 with five integers per line. *Hint*: Use the `%` operation.

2.2.8 Write a program that takes an integer `N` as a command-line argument and uses `Math.random` to print `N` uniform random values between 0 and 1 and then prints their average value (see Exercise 2.1.13).

2.2.9 Describe what happens when you invoke Program 2.2.4 with an argument that is too large, such as `java RulerN 100`.

2.2.10 Write a program `PowersOfK` that takes an integer `k` as command-line argument and prints all the positive powers of `k` in the Java `long` data type. *Note:* Recall that the constant `Long.MAX_VALUE` is the value of the largest integer in `long`.

2.2.11 Write a program `FunctionGrowth` that prints a table of the values of $\log N$, N, $N \log N$, N^2, N^3, and 2^N, for $N = 16, 32, 64, \ldots, 2048$. Use tabs (`\t` characters) to line up columns.

2.2.12 What does the following program print ?

```
int f = 0, g = 1;
for (int i = 0; i <= 15; i++)
  {
    System.out.println(f);
    f = f + g;
    g = f - g;
  }
```

Solution. Even an expert programmer will tell you that the only way to understand a program like this is to trace it. When you do, you will find that it prints the values 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 134, 233, 377, and 610 . These numbers are first sixteen of the famous *Fibonacci numbers*, which are defined by the following formulas: $F_0 = 0$, $F_1 = 1$, and $F_N = F_{N-1} + F_{N-2}$ for $N > 1$. The Fibonacci numbers arise in a surprising variety of contexts, they have been studied for centuries, and many of their properties are well-known. For example, the ratio of successive numbers approaches the golden ratio ϕ (about 1.618) as $N \rightarrow \infty$.

2.2.13 Write a version of the program in the previous exercise that prints the ratio of successive Fibonacci numbers, for all those in Java's `int` data type.

2.2.14 Expand your solution to Exercise 2.1.15 to print a table giving the total amount paid and the remaining principal after each monthly payment.

2.2.15 What is the value of `M` after executing the following code ?

```
int N = 123456789;
int M = 0;
while (N != 0)
  {
    M = (10 * M) + (N % 10);
    N = N / 10;
  }
```

2.2.16 The sum

$$\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \ldots + \frac{1}{N^2}$$

does converge to a constant as $N \rightarrow \infty$. (Indeed, the constant is $\pi^2/6$, so this formula can be used to compute the value of π.) Which of the following `for` loops computes this constant? Assume that `N` is an `int` initialized to `1000000` and `sum` is a `double` initialized to `0`.

a
```
for (int i = 1; i <= N; i++)
  sum += 1 / (i * i);
```

b
```
for (int i = 1; i <= N; i++)
  sum += 1.0 / i * i;
```

c
```
for (int i = 1; i <= N; i++)
  sum += 1.0 / (i * i);
```

d
```
for (int i = 1; i <= N; i++)
  sum += 1 / (1.0 * i * i);
```

2.2.17 Using the fact that the slope of the tangent to a (differentiable) function $f(x)$ at $x = t$ is $f'(t)$, find the equation of the tangent line and use that equation to find the point where the tangent line intersects the x axis to show that you can use Newton's method to find a root of any function $f(x)$ as follows: at each iteration, replace the estimate t by $t - f(t)/f'(t)$. Then use the fact that $f'(t) = 2t$ for $f(x) = x^2 - c$ to show that Program 2.2.6 implements Newton's method for finding the square root of *c*.

2.2.18 Use the general formula for Newton's method in Exercise 2.2.17 to develop a program `Root` that takes an integer `k` its second command-line argument and prints the `k`th root of the first command-line argument.

2.2.19 Suppose that `x` and `t` are variables of type `double` and `N` is a variable of type `int`. Write a code fragment to set `t` to $x^N/N!$.

Solution. A direct solution is to use one loop for the numerator and another loop for the denominator, then divide the results:

```
double num, dem;
for (i = 1, num = 1.0; i <= N; i++) num *= x;
for (i = 1, den = 1.0; i <= N; i++) den *= i;
t = num/den;
```

A better approach is to use just a single `for` loop:

```
for (i = 1, t = 1.0; i <= N; i++) t *= x/i;
```

Besides being more compact and elegant, the latter solution is preferable because it avoid inaccuracies caused by computing with huge numbers. For example, the two-loop approach breaks down for values like $x = 10$ and $N = 100$ because 100! is too large to represent as a `double`.

2.2.20 Modify `Binary` to get a program `Kary` that takes a second command-line argument `k` and to convert the first argument to base `k`. Assume that the first argument is an integer in Java's long data type and that the second is an integer between 2 and 20. For bases greater than 10, use the letters `A` through `K` to represent the 11th through 20th digits, respectively.

2.2.21 Write a code fragment that puts the binary representation of an `int N` into a `String s`.

Solution. Java has a built-in method `Integer.toBinaryString(N)` for this job, but the point of the exercise is to see how such a method might be implemented. Working from Program 2.2.7, we get the solution

```
for (v = 1; v <= N/2; v *= 2) ;
for (s = ""; v > 0; v /= 2)
  if (N < v) { s += 0;         }
  else       { s += 1; N -= v; }
```

A simpler option is to work from right to left:

```
for (s = ""; N > 0; N /= 2)
  s = (N % 2) + s;
```

Both of these methods are worthy of study.

2.2.22 Write a version of `Gambler` that uses `while` loops instead of `for` loops.

2.2.23 Write a program `GamblerPlot` that traces a gambler's ruin simulation by printing a line after each bet that has one asterisk corresponding to each dollar held by the gambler.

2.2.24 Modify `Gambler` to take an extra command-line parameter that specifies the (fixed) probability that the gamber wins each bet. Use your program to try to learn how this probability affects the chance of winning and the expected number of bets.

2.2.25 Modify `Gambler` to take an extra command-line parameter that specifies the number of bets the gambler is willing to make, so that there are three possible ways for the game to end: the gambler wins, loses, or runs out of time. Add to the output to give the expected amount of money the gambler will have when the game ends. *Extra credit*: Use your program to plan your next trip to Monte Carlo.

2.2.26 Modify `Factors` to print just one copy each of the prime divisors.

2.2.27 Run quick experiments to determine the impact of using the termination condition `(i <= N/i)` instead of `(i < N)` in `Factor` in Program 2.2.9. For each method, find the largest `M` such that when you type in an `M` digit number the program is sure to finish within 10 seconds.

2.2.28 Write a program `GCD` that finds the greatest common divisor (gcd) of two integers using *Euclid's algorithm*, which is an iterative computation based on the following observation: If `x` is greater than `y`, then if `y` divides `x`, the gcd of `x` and `y` is `y`; otherwise the gcd of `x` and `y` is the same as the gcd of `x % y` and `y`.

2.2.29 Write a program `Checkerboard` that takes one command line argument `N` and uses a loop within a loop to print out a two-dimensional `N`-by-`N` checkboard pattern with alternating spaces and asterisks.

2.2.30 Write a program `Divisors` that takes one command-line parameter `N` and prints out an `N` by `N` table such that there is a `*` in row `i` and column `j` if the gcd of `i` and `j` is `1` (`i` and `j` are relatively prime), and a space otherwise (see Exercise 2.2.28).

Creative Exercises

2.2.31 *Ramanujan's taxi.* S. Ramanujan was an Indian mathematician who became famous for his intuition for numbers. When the English mathematician G. H. Hardy came to visit him in the hospital one day, Hardy remarked that the number of his taxi was 1729, a rather dull number. To which Ramanujan replied, "No, Hardy! No, Hardy! It is a very interesting number. It is the smallest number expressible as the sum of two cubes in two different ways." Verify this claim by writing a program `Ramanujan.java` that takes a command line argument `N` and prints out all integers less than or equal to `N` that can be expressed as the sum of two

cubes in two different ways—find distinct positive integers a, b, c, and d such that $a^3 + b^3 = c^3 + d^3$. Use four nested `for` loops. *Note:* We will study faster ways to solve this problem in Chapter 4.

2.2.32 *Checksum*. The International Standard Book Number (ISBN) is a 10-digit code that uniquely specifies a book. The rightmost digit is a checksum digit which can be uniquely determined from the other 9 digits from the condition that $d_1 + 2d_2 + 3d_3 + \ldots + 10d_{10}$ must be a multiple of 11 (here d_i denotes the ith digit from the right). The checksum digit d_1 can be any value from 0 to 10: the ISBN convention is to use the value `X` to denote 10. *Example*: the checksum digit corresponding to 020131452 is 5 since 5 is the only value of x between 0 and 10 for which $1 \cdot x + 2 \cdot 2 + 3 \cdot 5 + 4 \cdot 4 + 5 \cdot 1 + 6 \cdot 3 + 7 \cdot 1 + 8 \cdot 0 + 9 \cdot 2 + 10 \cdot 0$ is a multiple of 11. Write a program that takes a 9-digit integer as a command line argument, computes the checksum, and prints out the the ISBN number.

2.2.33 *Calendar*. Write a program `Calendar` that takes two command-line arguments `M` and `Y` and prints out the monthly calendar for the `M`th month of year `Y`. For example, your output for `Calendar 2 2009` should be

```
   February 2009
S  M Tu  W Th  F  S
1  2  3  4  5  6  7
8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
```

Hint: See Program 2.1.4 and Exercise 2.1.17.

2.2.34 *Counting primes.* Write a program `PrimeCounter` that takes one command line argument `N` and finds the number of primes less than `N`. Use it to print out the number of primes less than 10 million. *Note:* if you are not careful to make your program efficient, it may not finish in a reasonable amount of time!

2.2.35 *Dragon curves.* Write a program that takes an integer N as command-line parameter and prints the instructions for drawing a dragon curve of order N (see Exercise 2.1.18).

2.2.36 *Exponential function.* Assume that `x` and `sum` are variables of type `double`. Write a code fragment to use the Taylor series expansion to set the value of `sum` to

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

Solution. A direct solution is to use a `for` loop within a `for` loop, based on the solution to Exercise 2.2.19:

```
double term = 1.0, sum = 1.0;
for (N = 1; sum != sum + term; N++)
  { term = 1.0;
    for (i = 1; i <= N; i++) term *= x/i;
    sum += term;
  }
```

The number of times the loop iterates depends on the relative values of the next term and the accumulated sum. Once the value of the sum stops changing, we leave the loop. (This strategy is more efficient than using the termination condition `(term > 0)` because it avoids a significant number of iterations that do not change the value of the sum.) This code is effective, but it is inefficient, because the inner for loop recomputes all the values it computed on the previous iteration of the outer for loop. Instead, we can make use of the term $x^{N-1}/(N-1)!$ that was added in on the previous loop iteration and solve the problem with a single `for` loop:

```
double term = 1.0; sum = 1.0;
for (int N = 1, sum != sum + term; N++)
  { term *= x/N; sum += term; }
```

We always seek improvements of this sort in our programs.

2.2.37 *Trigonometric functions.* Write two programs `Sin` and `Cos` that compute $\sin x$ and $\cos x$ using their Taylor series expansions $\sin x = -x + x^3/3! - x^5/5! + \ldots$ and $\cos x = 1 + x^2/2! + x^4/4! + \ldots$.

2.2.38 *Experimental analysis.* Run experiments to determine the relative costs of `Math.Exp` and the following three methods from Exercise 2.2.35 for the problem of computing e^x: , the direct method with nested `for` loops, the improved method with a single `for` loop, and the latter with the termination condition `(term > 0)`. For each method, use trial-and-error

with a command-line argument to determine how many times your computer can perform the computation in 10 seconds.

2.2.39 *2D random walk*. A two-dimensional random walk simulates the behavior of a particle moving in a grid of points. At each step, the random walker moves north, south, east, or west with probability $1/4$, independent of previous moves. Write a class `RandomWalker` that takes a command line parameter N and estimates how long it will take a random walker to hit the boundary of a $2N$ by $2N$ square centered at the starting point.

2.2.40 *Game simulation.* In the 1970s game show *Let's Make a Deal*, a contestant is presented with three doors. Behind one of them is a valuable prize, behind the other two are gag gifts. After the contestant chooses a door, the host opens one of the other two doors (never revealing the prize, of course). The contestant is then given the opportunity to switch to the other unopened door. Should the contestant do so? Intuitively, it might seem that the contestant's initial choice door and the other unnopened door are equally likely to contain the prize, so there would be no incentive to switch. Write a program `MonteHall` to test this intuition by simulation. Your program should take a command-line parameter `N`, play the game `N` times using each of the two strategies (switch or do not switch) and print the chance of success for each of the two strategies.

2.2.41 *Chaos*. Write a program to study the following simple model for population growth, which might be applied to study fish in a pond, bacteria in a test tube, or any of a host of similar situations. We suppose that the population ranges from 0 (extinct) to 1 (maximum population that can be sustained). If the population at time t is x, then we suppose the population at time $t+1$ to be $rx(1-x)$, where the parameter r, sometimes known as the *fecundity parameter*, controls the rate of growth. Start with a small population, say $x = 0.01$, and study the result of iterating the model, for various values of r. For which values of r does the population stabilize at $x = 1 - 1/r$? Can you say anything about the population when r is 3.5? 3.8? 5?

2.3 Arrays

2.4 Functions (static methods)

2.5 Recursion

2.6 Input and Output

3 *Object-Oriented Programming*

4 *Fundamental ADTs*

5 *A Computing Machine*

6 *Building a Computer*

7 *Theory of Computation*

8 *Systems*

9 *Scientific Computation*

10 *Analysis of Algorithms*