1.1 Your First Program

In this section, our plan is to lead you into the 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 many of the other applications that you are accustomed to using (such as your word processor, email program, and internet browser). As with any application, you need to be sure that Java is properly installed on your computer. It comes preloaded on many computers, or you can download it easily. You also need a text editor and a terminal application. Your first task is to find the instructions for installing such a Java programming environment on your computer by visiting

http://www.cs.princeton.edu/IntroProgramming

We refer to this site as the booksite. It contains an extensive amount of supplementary information about the material in this book for your reference and use. You will find it useful to have your browser open to this site while programming.

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

- Create a program by typing it into a file named, say, MyCode.java.
- Compile it by typing javac MyCode.java in a terminal window.
- Run (or execute) it by typing java MyCode in the terminal window.

In the first step, you start with a blank screen and end with a sequence of typed characters on the screen, just as when you write an email message or a paper. Programmers use the term code to refer to program text and the term coding to refer to the act of creating and editing the code. In the second step, you use a system application that compiles your program (translates it into a form more suitable for the computer) and puts the result in a file named MyCode.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). Many systems have several different ways to create, compile, and execute programs. We choose the sequence described here because it is the simplest to describe and use for simple programs.

Creating a program. A Java program is nothing more than a sequence of characters, like a paragraph or a poem, stored in a file with a .java extension. To create one, therefore, you need only define that sequence of characters, in the same way as you do for email or any other computer application. You can use any text editor for this task, or you can use one of the more sophisticated program development environments described on the booksite. Such environments are overkill for the sorts of programs we consider in this book, but they are not difficult to use, have many useful features, and are widely used by professionals.

Compiling a program. At first, it might seem that Java is designed to be best understood by the computer. To the contrary, the language is designed to be best understood by the programmer (that's you). The computer's language is far more primitive than Java. A compiler is an application that translates a program from the Java language to a language more suitable for executing on the computer. The compiler 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 your Java compiler, type in a terminal window the javac command followed by the file name of the program you want to compile.

 Executing a program. Once you compile the program, you can run it. This is the exciting part, where your program takes control of your computer (within the constraints of what the Java system allows). It is perhaps more accurate to say that your computer follows your instructions. It is even more accurate to say that a part of the Java system known as the Java Virtual Machine (the JVM, for short) directs your computer to follow your instructions. To use the JVM to execute your program, type the java command followed by the program name in a terminal window.
### 1.1 Your First Program

**Program 1.1.1 Hello, World**

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

This code is a Java program that accomplishes a simple task. It is traditionally a beginner’s first program. The box below shows what happens when you compile and execute the program. The terminal application gives a command prompt (% in this book) and executes the commands that you type (javac and then java in the example below). The result in this case is that the program prints a message in the terminal window (the third line).

% java HelloWorld.java
% java HelloWorld
Hello, World

**Program 1.1.1 is an example of a complete Java program.** Its name is HelloWorld, which means that its code resides 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, but we will not define them until later in the book. Program 1.1.1 consists of a single class named HelloWorld that has a single method named main(). This method uses two other methods named System.out.println() and System.out.print() to do the job. (When referring to a method in the text, we use () after the name to distinguish it from other kinds of names.) Until Section 2.1, where we learn about classes that define multiple methods, all of our classes will have this same structure. For the time being, you can think of “class” as meaning “program.”

The first line of a method specifies its name and other information; the rest is a sequence of statements enclosed in braces and each followed by a semicolon. For the time being, you can think of “programming” as meaning “specifying a class name and a sequence of statements for its main() method.” In the next two sections, you will learn many different kinds of statements that you can use to make programs. For the moment, we will just use statements for printing to the terminal like the ones in HelloWorld.

When you type java followed by a class name in your terminal application, the system calls the main() method that you defined in that class, and executes its statements in order, one by one. Thus, typing java HelloWorld causes the system to call on the main() method in PROGRAM 1.1.1 and execute its two statements. The first statement calls on System.out.println() to print in the terminal window the message between the quotation marks, and the second statement calls on System.out.print() to terminate the line.

Since the 1970s, it has been a tradition that a beginning programmer’s first program should print “Hello, world”. So, you should type the code in PROGRAM 1.1.1 into a file, compile it, and execute it. By doing so, you will be following in the footsteps of countless others who have learned how to program. Also, you will be checking that you have a usable editor and terminal application. At first, accomplishing the task of printing something out in a terminal window might not seem very interesting; upon reflection, however, you will see that one of the most basic functions that we need from a program is its ability to tell us what it is doing.

For the time being, all our program code will be just like PROGRAM 1.1.1, except with a different sequence of statements in main(). Thus, you do not need to start with a blank page to write a program. Instead, you can:

- **Copy** HelloWorld.java into a new file having a new program name of your choice, followed by .java.
- **Replace** HelloWorld on the first line with the new program name.
- **Replace** the System.out.println() and System.out.print() statements with a different sequence of statements (each ending with a semicolon).

Your program is characterized by its sequence of statements and its name. Each Java program must reside in a file whose name matches the one after the word class on the first line, and it also must have a .java extension.
1.1 Your First Program

Input and Output  Typically, we want to provide input to our programs: data that they can process to produce a result. The simplest way to provide input data is illustrated in UseArgument (Program 1.1.2). Whenever UseArgument 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. After compiling the program once, we can run it for different command-line arguments and get different printed results. We will discuss in more detail the mechanism that we use to pass arguments to our programs later, in Section 2.1. In the meantime, you can use args[0] within your program’s body to represent the string that you type on the command line when it is executed, just as in UseArgument.

Errors. It is easy to blur the distinction among editing, compiling, and executing programs. You should keep them separate in your mind when you are learning to program, to better understand the effects of the errors that inevitably arise. You can find several examples of errors in the Q&A at the end of this section. You can fix or avoid most errors by carefully examining the program as you create it, the same way you fix spelling and grammatical errors when you compose an email message. Some errors, known as compile-time errors, are caught when you compile the program, because they prevent the compiler from doing the translation. Other errors, known as run-time errors, do not show up until you execute the program. In general, errors in programs, also commonly known as bugs, are the bane of a programmer’s existence; the error messages can be confusing or misleading, and the source of the error can be very hard to find. One of the first skills that you will learn is to identify errors; you will also learn to be sufficiently careful when coding, to avoid making many of them in the first place.
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 be programming in other languages in the future.

Q. Do I really have to type in the programs in the book to try them out? I believe that you ran them and that they produced the indicated output.
A. Everyone should type in and run HelloWorld. Your understanding will be greatly magnified if you also run java Argument, try it on various inputs, and modify it to test different ideas of your own. To save some typing, you can find all of the code in this book (and much more) on the booksite. This site also has information about installing and running Java on your computer, answers to selected exercises, web links, and other extra information that you may find useful or interesting.

Q. What is the meaning of the words public, static and void?
A. These keywords specify certain properties of main() that you will learn about later in the book. For the moment, we just include these keywords in the code (because they are required) but do not refer to them in the text.

Q. What is the meaning of the //, /*, and */ character sequences in the code?
A. They denote comments, which are ignored by the compiler. A comment is either text in between /* and */ or at the end of a line after //. As with most online code, the code on the booksite is liberally annotated with comments that explain what it does; we use fewer comments in code in this book because the accompanying text and figures provide the explanation.

Q. What are Java’s rules regarding tabs, spaces, and newline characters?
A. Such characters are known as whitespace characters. Java compilers consider all whitespace in program text to be equivalent. For example, we could write He1-

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


% javac Bad.java
Bad.java:4: ';' expected
  System.out.print("Hello, World");
^  
Bad.java:7: 'class' or 'interface' expected
}  
^  
Bad.java:8: 'class' or 'interface' expected
^  
3 errors

One way to get used to such messages is to intentionally introduce mistakes into a simple program and 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 use many arguments, though we normally use just a few. Note that the count starts at 0, so you refer to the first argument as args[0], the second one as args[1], the third one as args[2], and so forth.

Q. What Java methods are available for me to use?

A. There are literally thousands of them. We introduce them to you in a deliberate fashion (starting in the next section) to avoid overwhelming you with choices.

Q. When I ran UseArgument, I got a strange error message. What's the problem?

A. Most likely, you forgot to include a command-line argument:

% java UseArgument
Hi, Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
    at UseArgument.main(UseArgument.java:6)

The JVM is complaining that you ran the program but did not type an argument as promised. You will learn more details about array indices in Section 1.4. Remember this error message: you are likely to see it again. Even experienced programmers forget to type arguments on occasion.

1.1 Your First Program

Exercises

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

1.1.2 Describe what happens if you omit the following in HelloWorld.java:
   a. public
   b. static
   c. void
   d. args

1.1.3 Describe what happens if you misspell (by, say, omitting the second letter) the following in HelloWorld.java:
   a. public
   b. static
   c. void
   d. args

1.1.4 Describe what happens if you try to execute UseArgument with each of the following command lines:
   a. java UseArgument java
   b. java UseArgument 016%
   c. java UseArgument 1234
   d. java UseArgument.java Bob
   e. java UseArgument Alice Bob

1.1.5 Modify UseArgument.java to make a program UseThree.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 UseThree Alice Bob Carol gives Hi Carol, Bob, and Alice.
1.2 Built-in Types of Data

When programming in Java, you must always be aware of the type of data that your program is processing. The programs in Section 1.1 process strings of characters, many of the programs in this section process numbers, and we consider numerous other types later in the book. Understanding the distinctions among them is so important that we formally define the idea: a data type is a set of values and a set of operations defined on those values. You are familiar with various types of numbers, such as integers and real numbers, and with operations defined on them, such as addition and multiplication. In mathematics, we are accustomed to thinking of sets of numbers as being infinite; in computer programs we have to work with a finite number of possibilities. Each operation that we perform is well-defined only for the finite set of values in an associated data type.

There are eight primitive types of data in Java, mostly for different kinds of numbers. Of the eight primitive types, we most often use these: int for integers; double for real numbers; and boolean for true/false values. There are other types of data available in Java libraries; for example, the programs in Section 1.1 use the type String for strings of characters. Java treats this type differently from other types 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. For clarity, we refer to primitive types and String collectively as built-in types. For the time being, we concentrate on programs that are based on computing with built-in types. Later, you will learn about Java library data types and building your own data types. Indeed, programming in Java is often centered on building data types, as you shall see in Chapter 3.

After defining basic terms, we consider several sample programs and code fragments that illustrate the use of different types of data. These code fragments do not do much real computing, but you will soon see similar code in longer programs. Understanding data types (values and operations on them) is an essential step in beginning to program. It sets the stage for us to begin working with more intricate programs in the next section. Every program that you write will use code like the tiny fragments shown in this section.

---

### 1.2.1 String concatenation example

1.2.2 Integer multiplication and division

1.2.3 Quadratic formula

1.2.4 Loop with a counter

1.2.5 Casting to get a random integer

---

### Program in this section

---

#### Definitions

To talk about data types, we need to introduce some terminology. To do so, we start with the following code fragment:

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

The first line is a declaration that declares the names of three variables to be the identifiers `a`, `b`, and `c` and their type to be int. The next three lines are assignment statements that change the values of the variables, using the literals `1234` and `99`, and the expression `a + b`, with the end result that `c` has the value `1333`.

**Identifiers.** We use identifiers to name variables (and many other things) in Java. An identifier is a sequence of letters, digits, _, $, which is the first of which is not a digit. The sequences of characters `abc`, `Ab5`, and `a_b` are all legal Java identifiers, while `Ab5`, `Abc`, and `a+b` are not. Identifiers are case-sensitive, so `Ab`, `ab`, and `AB` are different names. You cannot use certain reserved words—such as `public`, `static`, `int`, `double`, and so forth—to name variables.

**Literals.** A literal is a source-code representation of a data-type value. We use strings of digits like `1234` or `99` to define `int` literal values, and 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`, and to specify a `String`, we use a sequence of characters enclosed in quotes, such as "Hello", "World". We will consider other kinds of literals as we consider each data type in more detail.

**Variables.** A variable is a name that we use to refer to a data-type value. We use variables to keep track of changing values as a computation unfolds. For example,
we use the variable \( n \) in many programs to count things. We create a variable in a declaration that specifies its type and gives it a name. We compute with it by using the name in an expression that uses operations defined for its type. Each variable always stores one of the permissible data-type values.

Declaration statements. A declaration statement associates a variable name with a type at compile time. Java requires us to use declarations to specify the names and types of variables. By doing so, we are being explicit about any computation that we are specifying. Java is said to be a strongly-typed language, because the Java compiler can check for consistency at compile time (for example, it does not permit us to add a String to a double). This situation is precisely analogous to making sure that quantities have the proper units in a scientific application (for example, it does not make sense to add a quantity measured in inches to another measured in pounds). Declarations can appear anywhere before a variable is first used—most often, we put them at the point of first use.

Assignment statements. An assignment statement associates a data-type value with a variable. When we write \( c = a + b \) in Java, we are not expressing mathematical equality, but are instead expressing an action: set the value of the variable \( c \) to be the value of \( a \) plus the value of \( b \). It is true that \( c \) is mathematically equal to \( a + b \) immediately after the assignment statement has been executed, but the point of the statement is to change the value of \( c \) (if necessary). The left-hand side of an assignment statement must be a single variable; the right-hand side can be an arbitrary expression that produces values of the type. For example, we can say

\[
\text{int d = b + 4*a*c;}
\]

In short, the meaning of \( = \) is decidedly not the same as in mathematical equations. For example, \( a = b \) is certainly not the same as \( b = a \), and while the value of \( c \) is the value of \( a \) plus the value of \( b \) after \( c = a + b \) has been executed, that may cease to be the case if subsequent statements change the values of any of the variables.

Initialization. In a simple declaration, the initial value of the variable is undefined. For economy, we can combine a declaration with an assignment statement to provide an initial value for the variable.

1.2 Built-in Types of Data

Tracing in variable values. As a final check on your understanding of the purpose of assignment statements, convince yourself that the following code exchanges the values of \( a \) and \( b \) (assume that \( a \) and \( b \) are int variables):

\[
\begin{align*}
\text{int } & t; \\
& a = a; \\
& b = t;
\end{align*}
\]

To do so, use a time-honored method of examining program behavior: study a table of the variable values after each statement (such a table is known as a trace).

Expressions. An expression is a literal, a variable, or a sequence of operations on literals and/or variables that produces a value. For primitive types, expressions look just like mathematical formulas, which are based on familiar symbols or operators that specify data-type operations to be performed on one or more operands. Each operand can be any expression. Most of the operators that we use are binary operators that take exactly two operands, such as \( x + 1 \) or \( y / 2 \). An expression that is enclosed in parentheses is another expression with the same value. For example, we can write \( 4 \times (x - 3) \) or \( 4 \times x - 12 \) on the right-hand side of an assignment statement and the compiler will understand what we mean.

Precedence. Such expressions are shorthand for specifying a sequence of computations: in what order should they be performed? Java has natural and well-defined precedence rules (see the booksite) that fully specify this order. For arithmetic operations, multiplication and division are performed before addition and subtraction, so that \( a - b \times c \) and \( a - (b \times c) \) represent the same sequence of operations. When arithmetic operators have the same precedence, the order is determined by left-associativity, so that \( a - b - c \) and \( (a - b) - c \) represent the same sequence of operations. You can use parentheses to override the rules, so you should not need to worry about the details of precedence for most of the programs that you write. (Some of the programs that you read might depend subtly on precedence rules, but we avoid such programs in this book.)

Converting strings to primitive values for command-line arguments. Java provides the library methods that we need to convert the strings that we type as
Elements of Programming

1.2 Built-In Types of Data

Characters and Strings. A character is an alphanumeric value, like the ones you type. There are 255 different possible character values, but we usually restrict attention to the ones that represent the usual letters, symbols, and whitespace characters such as the English alphabet, digits, and " " and " " characters. In the example "Wiley". The String data type is a sequence of characters, and String operations are performed on this sequence. A String value is a sequence of characters enclosed in quotes, for example: "Wiley" or "example String".

- String = "Wiley";
- String = "example String";
- String = "char";
- String = "String";
- String = "Java";
- String = "C#";
- String = "Python";
- String = "Ruby";
- String = "JavaScript";
- String = "Swift";
- String = "Go";
- String = "R";
- String = "Perl";
- String = "Scala";

These examples illustrate the String data type and its operations. String values are sequences of characters enclosed in quotes, and String operations are performed on sequences of characters. String values can be compared using the usual equality and inequality operators, such as == and !=. String values can also be concatenated using the + operator, for example: "Wiley" + " example" = "Wiley example".

The String data type is a sequence of characters enclosed in quotes, and String operations are performed on this sequence. String values can be compared using the usual equality and inequality operators, such as == and !=. String values can also be concatenated using the + operator, for example: "Wiley" + " example" = "Wiley example".

The String data type is a sequence of characters enclosed in quotes, and String operations are performed on this sequence. String values can be compared using the usual equality and inequality operators, such as == and !=. String values can also be concatenated using the + operator, for example: "Wiley" + " example" = "Wiley example".

The String data type is a sequence of characters enclosed in quotes, and String operations are performed on this sequence. String values can be compared using the usual equality and inequality operators, such as == and !=. String values can also be concatenated using the + operator, for example: "Wiley" + " example" = "Wiley example".
1.2 Built-in Types of Data

Integers An int is an integer (natural number) between \(-2^{31}\) and \(2^{31}-1\). These bounds derive from the fact that integers are represented in binary with 32 binary digits: there are \(2^{32}\) possible values. (The term binary digit is omnipresent in computer science, and we nearly always use the abbreviation bit: a bit is either 0 or 1.) The range of possible int values is asymmetric because zero is included with the positive values. See the book site for more details about number representation, but in the present context it suffices to know that an int is one of the finite sets of values in the range just given. Sequences of the characters 0 through 9, possibly with a plus or minus sign at the beginning (that, when interpreted as decimal numbers, fall within the defined range), are integer literal values. We use ints frequently because they naturally arise when implementing programs.

Standard arithmetic operators for addition/subtraction (+ and -), multiplication (*), division (/), and remainder (%) for the int data type are built into Java. These operators take two int operands and produce an int result, with one significant exception—division or remainder by zero is not allowed. These operations are defined just as in grade school (keeping in mind that all results must be integers): given two int values a and b, the value of a / b is the number of times b goes into a with the fractional part discarded, and the value of a % b is the remainder that you get when you divide a by b. For example, the value of 17 / 3 is 5, and the value of 17 % 3 is 2. The int results that we get from arithmetic operations are just what we expect, except that if the result is too large to fit into int’s 32-bit representation, then it will be truncated in a well-defined manner. This situation is known as overflow. In

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 + 3</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5 - 3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5 * 3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>5 / 3</td>
<td>1</td>
<td>no fractional part</td>
</tr>
<tr>
<td>5 % 3</td>
<td>2</td>
<td>remainder</td>
</tr>
<tr>
<td>1 / 0</td>
<td>run-time error</td>
<td></td>
</tr>
<tr>
<td>3 * 5 - 2</td>
<td>13</td>
<td>n has precedence</td>
</tr>
<tr>
<td>3 + 5 / 2</td>
<td>5</td>
<td>/ has precedence</td>
</tr>
<tr>
<td>3 - 5 % 2</td>
<td>-4</td>
<td>left associative</td>
</tr>
<tr>
<td>(3 - 5) / 2</td>
<td>-4</td>
<td>better style</td>
</tr>
<tr>
<td>3 - (5 - 2)</td>
<td>0</td>
<td>unambiguous</td>
</tr>
</tbody>
</table>

Typical int expressions

As just discussed, our most frequent use (by far) of the concatenation operator is to put together results of computation for output with System.out.print() and System.out.println(). For example, we could simplify `UseArgument` (Program 1.1.2) by replacing its three statements with this single statement:

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

We have considered the String type first precisely because we need it for output (and command-line input) in programs that process other types of data.
Program 1.2.2  Integer multiplication and division

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

Arithmetic for integers is built in to Java. Most of this code is devoted to the task of getting the values in and out; the actual arithmetic is in the simple statements in the middle of the program that assign values to p, q, and r.

In general, we have to take care that such a result is not misinterpreted by our code. For the moment, we will be computing with small numbers, so you do not have to worry about these boundary conditions.

Program 1.2.2 illustrates basic operations for manipulating integers, such as the use of expressions involving arithmetic operators. It also demonstrates the use of `Integer.parseInt()` to convert `String` values on the command line to `int` values, as well as the use of automatic type conversion to convert `int` values to `String` values for output.

1.2 Built-in Types of Data

Three other built-in types are different representations of integers in Java. 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 other types to save space. You can find a table with the maximum and minimum values for each type on the book site, or you can figure them out for yourself from the numbers of bits.

Floating-point 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. We use floating-point numbers to represent real numbers, but they are decidedly not the same as real numbers! There are infinitely many real numbers, but we can only represent a finite number of floating-points in any digital computer representation. Floating-point numbers do approximate real numbers sufficiently well that we can use them in applications, but we often need to cope with the fact that we cannot always do exact computations.

We can use a sequence 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 $6.022 \times 10^{23}$. 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`. Beyond the built-in operators, the `Java Math library` defines the square root, trigonometric

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.141 + .03</td>
<td>3.171</td>
</tr>
<tr>
<td>3.141 - .03</td>
<td>3.111</td>
</tr>
<tr>
<td>6.022e23 / 2</td>
<td>3.01e23</td>
</tr>
<tr>
<td>5.0 / 3.0</td>
<td>1.66666666666667</td>
</tr>
<tr>
<td>10.0 % 3.141</td>
<td>0.577</td>
</tr>
<tr>
<td>1.0 / 0.0</td>
<td>Infinity</td>
</tr>
<tr>
<td>Math.sqrt(2.0)</td>
<td>1.4142135623730951</td>
</tr>
<tr>
<td>Math.sqrt(-1.0)</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Typical double expressions

![Typical double expressions](image_url)

<table>
<thead>
<tr>
<th>values</th>
<th>approximations to real numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14159</td>
<td><code>6.022e23</code> $-3.0$</td>
</tr>
<tr>
<td>operations</td>
<td>operators</td>
</tr>
<tr>
<td>add</td>
<td>subtract</td>
</tr>
<tr>
<td>multiply</td>
<td>divide</td>
</tr>
<tr>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

Java's built-in double data type
1.2 Built-in Types of Data

ling the number of significant digits that you see in output. Until then, we will work
with the Java default output format.

The result of a calculation can be 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). Though there are myriad details to consider when calculations involve
these values, you can use double in a natural way and begin to write Java programs
instead of using a calculator for all kinds of calculations. For example, PROGRAM
1.2.3 shows the use of double values in computing the roots of a quadratic equa-
tion using the quadratic formula. Several of the exercises at the end of this section
further illustrate this point.

As with long, short, and byte for integers, there is another representation
for real numbers called float. Programmers sometimes use float 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 do not use
float in this book.

Booleans The boolean type has just two values: true and false. These are the two possible boolean literals. Ev-
ery 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 simplicity is deceiving—
boolean values lie at the foundation of computer science.

The most important operators defined for booleans are for the and (&&), or
(||), and not (!) operations, which have familiar definitions:

• a && b is true if both operands are true, and false if either is false.
• a || b is false if both operands are false, and true if either is true.
• !a is true if a is false, and false if a is true.

Despite the intuitive nature of these definitions, it is worthwhile to fully specify
each possibility for each operation in tables known as truth tables. The not function

<table>
<thead>
<tr>
<th>a</th>
<th>!a</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

Truth-table definitions of boolean operations
Elements of Programming

| a   | b   | a & b | !a | !b | !a || !b | !(a || !b) |
|-----|-----|------|----|----|-----|-----|----------|
| false | false | false | true | true | true | false |
| false | true  | false | true | false | false | false |
| true  | false | false | false | true | false | false |
| true  | true  | true  | false | false | false | true  |

Truth-table proof that a & b and !(a || !b) are identical

has only one operand: its value for each of the two possible values of the operand is specified in the second column. The 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. This field of mathematics is fundamental to computing: it plays an essential role in the design and operation of computer hardware itself, and it is also a starting point for the theoretical foundations of computation. 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 expression and a piece of program code is written to execute one set of statements if the expression is true and a different set of statements if the expression is false. The mechanics of doing so are the topic of Section 1.3.

Comparisons Some mixed-type operators take operands of one type and produce a result of another type. The most important operators of this kind are the comparison operators ==, !=, <, <=, >, and >=, which all are defined for each primitive numeric type and produce a boolean result. Since operations are defined only for

non-negative discriminant? (b*b - 4.0*a*c) >= 0.0
beginning of a century? (year % 100) == 0
legal month? (month >= 1) & & (month <= 12)

Typical comparison expressions

1.2.4 Built-in Types of Data

Program 1.2.4 Leap year

```java
public class LeapYear {
    public static void main(String[] args) {
        int year = Integer.parseInt(args[0]);
        boolean isLeapYear;
        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).

% Java LeapYear.java
% Java LeapYear 2004
true
% Java LeapYear 1900
false
% Java LeapYear 2000
true

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. The result is always boolean.

Even without going into the details of number representation, it is clear that the operations for the various types are really quite different: for example, it is one thing to compare two ints to check that (2 <= 2) is true but quite another to compare two doubles to check whether (2.0 <= 0.002e3) is true or false. Still, these operations are well-defined and useful to write code that tests for conditions such as (b*b - 4.0*a*c) >= 0.0, which is frequently needed, as you will see.
Elements of Programming

The comparison operations have lower precedence than arithmetic operators and higher precedence than boolean operators, so you do not need the parentheses in an expression like \((b+b - 4.0*a*c) >= 0.0\), and you could write an expression like `month >= 1 && month <= 12` without parentheses to test whether the value of the `int` variable `month` is between 1 and 12. (It is better style to use the parentheses, however.)

Comparison operations, together with boolean logic, provide the basis for decision-making in Java programs. Program 1.2.4 is an example of their use, and you can find other examples in the exercises at the end of this section. More importantly, in Section 1.3 we will see the role that boolean expressions play in more sophisticated programs.

Library methods and APIs As we have seen, many programming tasks involve using Java library methods in addition to the built-in operations on data-type values. The number of available library methods is vast. As you learn to program, you will learn to use more and more library methods, but it is best at the beginning to restrict your attention to a relatively small set of methods. In this chapter, you have already used some of Java's methods for printing, for converting data from one type to another, and for computing mathematical functions (the Java Math library). In later chapters, you will learn not just how to use other methods, but how to create and use your own methods.

For convenience, we will consistently summarize the library methods that you need to know how to use in tables like this one:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Math.abs(double a)</code></td>
<td>absolute value of <code>a</code></td>
</tr>
<tr>
<td><code>Math.max(double a, double b)</code></td>
<td>maximum of <code>a</code> and <code>b</code></td>
</tr>
<tr>
<td><code>Math.min(double a, double b)</code></td>
<td>minimum of <code>a</code> and <code>b</code></td>
</tr>
<tr>
<td><code>Math.sin(double theta)</code></td>
<td>sine function</td>
</tr>
<tr>
<td><code>Math.cos(double theta)</code></td>
<td>cosine function</td>
</tr>
<tr>
<td><code>Math.tan(double theta)</code></td>
<td>tangent function</td>
</tr>
<tr>
<td><code>Math.exp(double a)</code></td>
<td>exponential (e^a)</td>
</tr>
<tr>
<td><code>Math.log(double a)</code></td>
<td>natural log ((\log_2 a), or (\ln a))</td>
</tr>
<tr>
<td><code>Math.pow(double a, double b)</code></td>
<td>raise <code>a</code> to the <code>b</code>th power ((a^b))</td>
</tr>
<tr>
<td><code>Math.round(double a)</code></td>
<td>round the nearest integer</td>
</tr>
<tr>
<td><code>Math.random()</code></td>
<td>random number in ([0, 1])</td>
</tr>
<tr>
<td><code>Math.sqrt(double a)</code></td>
<td>square root of <code>a</code></td>
</tr>
<tr>
<td><code>Math.E</code></td>
<td>value of (e) (constant)</td>
</tr>
<tr>
<td><code>Math.PI</code></td>
<td>value of (\pi) (constant)</td>
</tr>
</tbody>
</table>

See booksite for other available functions.

Excerpts from Java's mathematics library

With the exception of `random()`, these methods implement mathematical functions—they use their arguments to compute a value of a specified type. Each method is described by a line in the API that specifies the information you need to know in order to use the method. The code in the tables is not the code that you type to use the method; it is known as the method's signature. The signature specifies the type of the arguments, the method name, and the type of the value that the method computes (the return value). When your program is executed, we say that it calls the system library code for the method, which returns the value for use in your code.
Elements of Programming

1.2 Built-in Types of Data

<table>
<thead>
<tr>
<th>expression</th>
<th>library</th>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer.parseInt(&quot;123&quot;)</td>
<td>Integer</td>
<td>int</td>
<td>123</td>
</tr>
<tr>
<td>Math.sqrt(5.0<em>5.0 - 4.0</em>4.0)</td>
<td>Math</td>
<td>double</td>
<td>3.0</td>
</tr>
<tr>
<td>Math.random()</td>
<td>Math</td>
<td>double</td>
<td>random in [0,1]</td>
</tr>
<tr>
<td>Math.round(3.14159)</td>
<td>Math</td>
<td>long</td>
<td>3</td>
</tr>
</tbody>
</table>

Typical expressions that use Java library methods

These APIs are typical of the online documentation that is the standard in modern programming. There is extensive online documentation of the Java APIs that is used by professional programmers, and it is available to you (if you are interested) directly from the Java website or through our bookself. You do not need to go to the online documentation to understand the code in this book or to write similar code, because we present and explain in the text all of the library methods that we use in APIs like these and summarize them in the endpapers. More important, in Chapters 2 and 3 you will learn in this book how to develop your own APIs and to implement functions for your own use.

Type conversion One of the primary rules of modern programming 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 which set of values each variable can have, which literals you can use, and which operations you can perform. Typical programming tasks involve processing multiple types of data, so we often need 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 produces a result of another type. We have already used the Integer.parseInt() and Double.parseDouble() library methods to convert String values to int and double values, respectively. Many other methods are available for conversion among other types. For example, the library method Math.round() takes a double argument and returns a 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).

Explicit cast. Java has some built-in type conversion conventions for primitive types that you can take advantage of when you are aware that you might lose infor-
Elements of Programming

1.2 Built-in Types of Data

Program 1.2.5 Casting to get a random integer

```java
public class RandomInt {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        double r = Math.random(); // uniform between 0 and 1
        int n = (int) (r * N); // uniform between 0 and N-1
        System.out.println(n);
    }
}
```

This program uses the Java method Math.random() to generate a random number r in the interval [0, 1), then multiplies r by the command-line argument N to get a random number greater than or equal to 0 and less than N, then uses a cast to truncate the result to be an integer n between 0 and N-1.

Casting has higher precedence than arithmetic operations—any cast is applied to the value that immediately follows it. For example, if we write int n = (int) 11 * 0.3, the cast is no help: the literal 11 is already an integer, so the cast (int) has no effect. In this example, the compiler produces a possible loss of precision error message because there would be a loss of precision in converting the resulting value (3.3) to an int for assignment to n. The error is helpful because the intended computation for this code is likely (int) (11 * 0.3), which has the value 3, not 3.3.

Automatic promotion for numbers. You can use data of any primitive numeric type where a value whose type has a larger range of values is expected, because Java automatically converts to the type with the larger range. This kind of conversion is called promotion. For example, we used numbers all of type double in Program 1.2.3, so there is no conversion. If we had chosen to make a and c of type int (using Integer.parseInt() to convert the command-line arguments), automatic promotion would be used to evaluate the expression b*b - 4.0*c. First, c is promoted to double to multiply by the double literal 4.0, with a double result. Then, the int value b*b is promoted to double for the subtraction, leaving a double result. Or, we might have written b*b - 4*c. In that case, the expression b*b - 4*c would be evaluated as an int and then the result promoted to double, because that is what Math.sqrt() expects. Promotion is appropriate because your intent is clear and it can be done with no loss of information. On the other hand, a conversion that might involve loss of information (for example, assigning a double to an int) leads to a compile-time error.
BEGINNING PROGRAMMERS TEND TO FIND TYPE conversion to be an annoyance, but experienced programmers know that paying careful attention to data types is a key to success in programming. It is well worth your while to take the time to understand what type conversion is all about. After you have written just a few programs, you will understand that these rules help you to make your intentions explicit and to avoid subtle bugs in your programs.

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. In Java code, we use operators and expressions like those in familiar mathematical expressions to invoke the operations associated with each type. 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. In this book, we most often use boolean, int, and double; we do not use short or float. Another data type that we use frequently, String, is not primitive, but Java has some built-in facilities for Strings that are like those for primitive types.

When programming in Java, we have to be aware that every operation is defined only in the context of its data type (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 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 >=. Specifically, we use boolean expressions to control Java's conditional (if) and loop (for and while) constructs, which we will study in detail in the next section.

The numeric types and Java's libraries give us the ability to use Java as an extensive mathematical calculator. We write arithmetic expressions using the built-in operators +, -, * /, and % along with Java methods from the Math library. Although the programs in this section are quite rudimentary 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 and basic mathematical functions extensively in Java programming, so the effort that you spend now understanding them will certainly be worthwhile.

### Q&A

Q. What happens if I forget to declare a variable?

A. The compiler complains, as shown below for a program IntOpsBad, which is the same as Program 1.2.2 except that the int variable p is omitted from the declaration statement.

```java
% 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 will get quite a few error messages. A good strategy is to correct the first error and check that correction before addressing later ones.

Q. What happens if I forget to initialize a variable?

A. The compiler checks for this condition and will give you a variable might not have been initialized error message if you try to use the variable in an expression.

Q. Is there a difference between = and ==?

A. Yes, they are quite different! The first is an assignment operator that changes the value of a variable, and the second is a comparison operator that produces a boolean result. Your ability to understand this answer is a sure test of whether you understood the material in this section. Think about how you might explain the difference to a friend.
Q. Why do int values sometime become negative when they get large?

A. If you have not experienced this phenomenon, see Exercise 1.2.10. The problem has to do with the way integers are represented in the computer. You can learn the details on the bookstore. In the meantime, a safe strategy is using the int type when you know the values to be less than ten digits and the long type when you think the values might get to be ten digits or more.

Q. It seems wrong that Java should just let ints overflow and give bad values. Shouldn't Java automatically check for overflow?

A. Yes, this issue is a contentious one among programmers. The short answer for now is that the lack of such checking is one reason such types are called primitive data types. A little knowledge can go a long way in avoiding such problems. Again, it is fine to use the int type for small numbers, but when values run into the billions, you cannot.

Q. What is the value of Math.abs(-2147483648)?

A. -2147483648. This strange (but true) result is a typical example of the effects integer overflow.

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. The Java library function System.out.println() is one way to do the job, and it is similar to the basic printing method in the C programming language and many modern languages, as discussed in Section 1.5. Until then, 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 I initialize a double variable to infinity?

A. Java has built-in constants available for this purpose: Double.POSITIVE_INFINITY and Double.NEGATIVE_INFINITY.

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 remember that 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) before performing the comparison.

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

A. No and yes. The first generates a run-time exception for division by zero (which stops your program because the value is undefined); the second is legal and has the value infinity.

Q. Are there functions in Java's Math library for other trigonometric functions, like cosecant, secant, and cotangent?

A. No, because you could use Math.sin(), Math.cos(), and Math.tan() to compute them. Choosing which functions to include in an API is a tradeoff between the convenience of having every function that you need and the annoyance of having to find one of the few that you need in a long list. No choice will satisfy all users, and the Java designers have many users to satisfy. Note that there are plenty of redundancies even in the APIs that we have listed. For example, you could use Math.sin(x)/Math.cos(x) instead of Math.tan(x).

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

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

Q. How about == and !=?

A. Yes, but the result may not be what you expect, because of the meanings these operators have for non-primitive types. For example, there is a distinction between
1.2 Built-in Types of Data

Exercises

1.2.1 Suppose that a and b are int values. What does the following sequence of statements do?

```java
int t = a; b = t; a = b;
```

1.2.2 Write a program that uses Math.sin() and Math.cos() to check that the value of \( \cos^2 \theta + \sin^2 \theta \) is approximately 1 for any \( \theta \) entered as a command-line argument. Just print the value. Why are the values not always exactly 1?

1.2.3 Suppose that a and b are int values. Show that the expression

```java
((a & b) & & (a | | b)) || ((a & b) | | (a | | b))
```

is equivalent to true.

1.2.4 Suppose that a and b are int values. Simplify the following expression:

```java
(!((a < b) & & !(a > b)))
```

1.2.5 The exclusive or operator ^ for boolean operands is defined to be true if they are different, false if they are the same. Give a truth table for this function.

1.2.6 Why does 10/3 give 3 and not 3.333333333?

**Solution.** Since both 10 and 3 are integer literals, Java sees no 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.

1.2.7 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);`

Explain each outcome.

---

a String and its value. The expression "abc" == "ab" + x is false when x is a String with value "c" because the two operands are stored in different places in memory (even though they have the same value). This distinction is essential, as you will learn when we discuss it in more detail in Section 3.1.

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

A. The quotient \( a / b \) rounds toward 0; the remainder \( a \mod b \) is defined such that \( (a / b) \times b + a \mod b \) is always equal to a. For example, \(-14/3\) and \(14/-3\) are both -4, but \(-14 \mod 3\) is -2 and \(14 \mod -3\) is 2.

Q. Will \( a < b < c \) test whether three numbers are in order?

A. No, that will not compile. You need to say \( \langle a < b \& \& b < c \rangle \).

Q. Fifteen digits for floating-point numbers 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 precision, whereas the computer always deals with approximations. For example, \(0.1 + 0.1 = 0.2\) is true but \(0.1 + 0.1 + 0.1 = 0.3\) is false! Pitfalls like this are not at all unusual in scientific computing. Novice programmers should avoid comparing two floating-point numbers for equality.

Q. Why do we say \( a \& b \) and not \( a \& b \)?

A. Java also has a & operator that we do not use in this book but which you may encounter if you pursue advanced programming courses.

Q. Why is the value of 10^6 not 100000 but 12?

A. The ^ operator is not an exponentiation operator, which you must have been thinking. Instead, it is an operator like & that we do not use in this book. You want the literal 1e6. You could also use Math.pow(10, 6) but doing so is wasteful if you are raising 10 to a known power.
1.2.8 Explain how to use Program 1.2.3 to find the square root of a number.

1.2.9 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));

   Explain each outcome.

1.2.10 Suppose that a variable \( a \) is declared as int \( a = 2147483647 \) (or equivalently, Integer.MAX_VALUE). 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.

1.2.11 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(8/a);
   e. System.out.println((int) (8/a));

   Explain each outcome.

1.2.12 Describe what happens if you write Math.sqrt instead of Math.sqrt in Program 1.2.3.

1.2.13 What is the value of \( (\text{Math.sqrt}(2) * \text{Math.sqrt}(2) == 2) \) ?

1.2.14 Write a program that takes two positive integers as command-line arguments and prints true if either evenly divides the other.

1.2.15 Write a program that takes three positive integers as command-line arguments and prints true if any one of them is greater than or equal to the sum of the other two and false otherwise. \( \text{(Note: This computation tests whether the three numbers could be the lengths of the sides of some triangle.)} \)

1.2.16 A physics student gets unexpected results when using the code
   \[
   F = G * \text{mass1} * \text{mass2} / r^2
   \]
   to compute values according to the formula \( F = Gm_1m_2 / r^2 \). Explain the problem and correct the code.

1.2.17 Give the value of a after the execution of each of the following sequences:
   \[
   \begin{align*}
   &\text{int } a = 1; \quad \text{boolean } a = true; \quad \text{int } a = 2; \\
   &a = a + a; \quad a = 1a; \quad a = a * a; \\
   &a = a + a; \quad a = 1a; \quad a = a * a;
   \end{align*}
   \]

1.2.18 Suppose that \( x \) and \( y \) are double values that represent the Cartesian coordinates of a point \((x, y)\) in the plane. Give an expression whose value is the distance of the point from the origin.

1.2.19 Write a program that takes two int values \( a \) and \( b \) from the command line and prints a random integer between \( a \) and \( b \).

1.2.20 Write a program that prints the sum of two random integers between 1 and 6 (such as you might get when rolling dice).

1.2.21 Write a program that takes a double value \( t \) from the command line and prints the value of \( \sin(2t) + \sin(3t) \).

1.2.22 Write a program that takes three double values \( x_0, y_0, \) and \( t \) from the command line and prints the value of \( x_t + y_t + g^t/2, \) where \( g \) is the constant 9.87033. \( \text{(Note: This value the displacement in meters after } t \text{ seconds when an object is thrown straight up from initial position } x_0 \text{ at velocity } y_0 \text{ meters per second.)} \)

1.2.23 Write a program that takes two int values \( m \) and \( d \) from the command line and prints true if \( d \) of month \( m \) is between 3/20 and 6/20, false otherwise.