

CHAPTER 4

FUNDAMENTAL DATA TYPES



CHAPTER GOALS

- To understand integer and floating-point numbers
- To recognize the limitations of the numeric types
- To become aware of causes for overflow and roundoff errors
- To understand the proper use of constants
- To write arithmetic expressions in Java
- To use the String type to manipulate character strings
- To write programs that read input and produce formatted output

CHAPTER CONTENTS

4.1 NUMBERS 132

Syntax 4.1: Constant Declaration 136

Special Topic 4.1: Big Numbers 138

Programming Tip 4.1: Do Not Use Magic Numbers 139

4.2 ARITHMETIC 139

Syntax 4.2: Cast 143

Common Error 4.1: Unintended Integer Division 144

Common Error 4.2: Unbalanced Parentheses 144

Programming Tip 4.2: Spaces in Expressions 145

Special Topic 4.2: Combining Assignment and Arithmetic 145

Special Topic 4.3: Instance Methods and Static Methods 145

Computing & Society 4.1: The Pentium Floating-Point Bug 146

4.3 INPUT AND OUTPUT 147

Syntax 4.3: Input Statement 147

How To 4.1: Carrying Out Computations 151

Worked Example 4.1: Computing the Volume and Surface Area of a Pyramid 🌐

4.4 PROBLEM SOLVING: FIRST DO IT BY HAND 154

Worked Example 4.2: Computing Travel Time 🌐

4.5 STRINGS 156

Programming Tip 4.3: Reading Exception Reports 162

Special Topic 4.4: Using Dialog Boxes for Input and Output 162

Computing & Society 4.2: International Alphabets and Unicode 163



Numbers and character strings (such as the ones on this display board) are important data types in any Java program. In this chapter, you will learn how to work with numbers and text, and how to write simple programs that perform useful tasks with them. We also cover the important topic of input and output, which enables you to implement interactive programs.

4.1 Numbers

We start this chapter with information about numbers. The following sections tell you how to choose the most appropriate number types for your numeric values, and how to work with constants—numeric values that do not change.

4.1.1 Number Types

Java has eight primitive types, including four integer types and two floating-point types.

In Java, every value is either a reference to an object, or it belongs to one of the eight **primitive types** shown in Table 1.



Six of the primitive types are number types; four of them for integers and two for floating-point numbers.

Each of the number types has a different range. Appendix G explains why the range limits are related to powers of two. The largest number that can be represented in an `int` is denoted by `Integer.MAX_VALUE`. Its value is about 2.14 billion. Similarly, the smallest integer is `Integer.MIN_VALUE`, about -2.14 billion.

Table 1 Primitive Types

Type	Description	Size
<code>int</code>	The integer type, with range $-2,147,483,648$ (<code>Integer.MIN_VALUE</code>) . . . $2,147,483,647$ (<code>Integer.MAX_VALUE</code> , about 2.14 billion)	4 bytes
<code>byte</code>	The type describing a single byte, with range -128 . . . 127	1 byte
<code>short</code>	The short integer type, with range $-32,768$. . . $32,767$	2 bytes
<code>long</code>	The long integer type, with range $-9,223,372,036,854,775,808$. . . $9,223,372,036,854,775,807$	8 bytes
<code>double</code>	The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits	8 bytes
<code>float</code>	The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits	4 bytes
<code>char</code>	The character type, representing code units in the Unicode encoding scheme (see Computing & Society 4.2 on page 163)	2 bytes
<code>boolean</code>	The type with the two truth values <code>false</code> and <code>true</code> (see Chapter 5)	1 bit

Table 2 Number Literals in Java

Number	Type	Comment
6	int	An integer has no fractional part.
-6	int	Integers can be negative.
0	int	Zero is an integer.
0.5	double	A number with a fractional part has type double.
1.0	double	An integer with a fractional part .0 has type double.
1E6	double	A number in exponential notation: 1×10^6 or 1000000. Numbers in exponential notation always have type double.
2.96E-2	double	Negative exponent: $2.96 \times 10^{-2} = 2.96 / 100 = 0.0296$
 100,000		Error: Do not use a comma as a decimal separator.
 3 1/2		Error: Do not use fractions; use decimal notation: 3.5

When a value such as 6 or 0.335 occurs in a Java program, it is called a **number literal**. If a number literal has a decimal point, it is a floating-point number; otherwise, it is an integer. Table 2 shows how to write integer and floating-point literals in Java.

Generally, you will use the `int` type for integer quantities. Occasionally, however, calculations involving integers can *overflow*. This happens if the result of a computation exceeds the range for the number type. For example,

```
int n = 1000000;
System.out.println(n * n); // Prints -727379968, which is clearly wrong
```

The product $n * n$ is 10^{12} , which is larger than the largest integer (about $2 \cdot 10^9$). The result is truncated to fit into an `int`, yielding a value that is completely wrong. Unfortunately, there is no warning when an integer overflow occurs.

If you run into this problem, the simplest remedy is to use the `long` type. Special Topic 4.1 on page 138 shows you how to use the `BigInteger` type in the unlikely event that even the `long` type overflows.

Overflow is not usually a problem for double-precision floating-point numbers. The `double` type has a range of about $\pm 10^{308}$. Floating-point numbers have a different problem—limited precision. The `double` type has about 15 significant digits, and there are many numbers that cannot be accurately represented as `double` values.

When a value cannot be represented exactly, it is rounded to the nearest match. Consider this example:

```
double f = 4.35;
System.out.println(100 * f); // Prints 434.99999999999994
```

A numeric computation overflows if the result falls outside the range for the number type.

Rounding errors occur when an exact representation of a floating-point number is not possible.

If a computation yields an integer that is larger than the largest int value (about 2.14 billion), it overflows.



Floating-point numbers have limited precision. Not every value can be represented precisely, and roundoff errors can occur.



The problem arises because computers represent numbers in the binary number system. In the binary number system, there is no exact representation of the fraction $1/10$, just as there is no exact representation of the fraction $1/3 = 0.33333$ in the decimal number system. (See Appendix G for more information.)

For this reason, the `double` type is not appropriate for financial calculations. In this book, we will continue to use `double` values for bank balances and other financial quantities so that we keep our programs as simple as possible. However, professional programs need to use the `BigDecimal` type for this purpose—see Special Topic 4.1.

In Java, it is legal to assign an integer value to a floating-point variable:

```
int dollars = 100;
double balance = dollars; // OK
```

But the opposite assignment is an error: You cannot assign a floating-point expression to an integer variable.

```
double balance = 13.75;
int dollars = balance; // Error
```

You will see in Section 4.2.5 how to convert a value of type `double` into an integer.

In this book, we do not use the `float` type. It has less than 7 significant digits, which greatly increases the risk of **roundoff errors**. Some programmers use `float` to save on memory if they need to store a huge set of numbers that do not require much precision.

4.1.2 Constants

In many programs, you need to use numerical **constants**—values that do not change and that have a special significance for a computation.

A typical example for the use of constants is a computation that involves coin values, such as the following:

```
payment = dollars + quarters * 0.25 + dimes * 0.1
          + nickels * 0.05 + pennies * 0.01;
```

Most of the code is self-documenting. However, the four numeric quantities, 0.25, 0.1, 0.05, and 0.01 are included in the arithmetic expression without any explanation. Of course, in this case, you know that the value of a nickel is five cents, which explains the 0.05, and so on. However, the next person who needs to maintain this code may live in another country and may not know that a nickel is worth five cents.

Thus, it is a good idea to use symbolic names for all values, even those that appear obvious. Here is a clearer version of the computation of the total:

```
double quarterValue = 0.25;
double dimeValue = 0.1;
double nickelValue = 0.05;
double pennyValue = 0.01;
```

```
payment = dollars + quarters * quarterValue + dimes * dimeValue
        + nickels * nickelValue + pennies * pennyValue;
```

There is another improvement we can make. There is a difference between the `nickels` and `nickelValue` variables. The `nickels` variable can truly vary over the life of the program, as we calculate different payments. But `nickelValue` is always 0.05.

A `final` variable is a constant. Once its value has been set, it cannot be changed.

In Java, constants are identified with the reserved word `final`. A variable tagged as `final` can never change after it has been set. If you try to change the value of a `final` variable, the compiler will report an error and your program will not compile.

Many programmers use all-uppercase names for constants (`final` variables), such as `NICKEL_VALUE`. That way, it is easy to distinguish between variables (with mostly lowercase letters) and constants. We will follow this convention in this book. However, this rule is a matter of good style, not a requirement of the Java language. The compiler will not complain if you give a `final` variable a name with lowercase letters.

Here is an improved version of the code that computes the value of a payment.

```
final double QUARTER_VALUE = 0.25;
final double DIME_VALUE = 0.1;
final double NICKEL_VALUE = 0.05;
final double PENNY_VALUE = 0.01;
payment = dollars + quarters * QUARTER_VALUE + dimes * DIME_VALUE
        + nickels * NICKEL_VALUE + pennies * PENNY_VALUE;
```

Use named constants to make your programs easier to read and maintain.

Frequently, constant values are needed in several methods. Then you should declare them together with the instance variables of a class and tag them as `static` and `final`. As before, `final` indicates that the value is a constant. The `static` reserved word means that the constant belongs to the class—this is explained in greater detail in Chapter 8.)

```
public class CashRegister
{
    // Constants
    public static final double QUARTER_VALUE = 0.25;
    public static final double DIME_VALUE = 0.1;
    public static final double NICKEL_VALUE = 0.05;
    public static final double PENNY_VALUE = 0.01;

    // Instance variables
    private double purchase;
    private double payment;

    // Methods
    . . .
}
```

We declared the constants as `public`. There is no danger in doing this because constants cannot be modified. Methods of other classes can access a public constant by first specifying the name of the class in which it is declared, then a period, then the name of the constant, such as `CashRegister.NICKEL_VALUE`.

The `Math` class from the standard library declares a couple of useful constants:

```
public class Math
{
    . . .
    public static final double E = 2.7182818284590452354;
    public static final double PI = 3.14159265358979323846;
}
```

You can refer to these constants as `Math.PI` and `Math.E` in any method. For example,

```
double circumference = Math.PI * diameter;
```

Syntax 4.1 Constant Declaration

Syntax Declared in a method: `final typeName variableName = expression;`
 Declared in a class: `accessSpecifier static final typeName variableName = expression;`

Declared in a method

`final double NICKEL_VALUE = 0.05;`

The `final` reserved word indicates that this value cannot be modified.

Use uppercase letters for constants.

`public static final double LITERS_PER_GALLON = 3.785;`

Declared in a class

The sample program below puts constants to work. The program shows a refinement of the `CashRegister` class of How To 3.1. The public interface of that class has been modified in order to solve a common business problem.

Busy cashiers sometimes make mistakes totaling up coin values. Our `CashRegister` class features a method whose inputs are the *coin counts*. For example, the call

```
register.receivePayment(1, 2, 1, 1, 4);
```

processes a payment consisting of one dollar, two quarters, one dime, one nickel, and four pennies. The `receivePayment` method figures out the total value of the payment, \$1.69. As you can see from the code listing, the method uses named constants for the coin values.

section_1/CashRegister.java

```

1  /**
2   * A cash register totals up sales and computes change due.
3   */
4  public class CashRegister
5  {
6      public static final double QUARTER_VALUE = 0.25;
7      public static final double DIME_VALUE = 0.1;
8      public static final double NICKEL_VALUE = 0.05;
9      public static final double PENNY_VALUE = 0.01;
10
11     private double purchase;
12     private double payment;
13
14     /**
15      * Constructs a cash register with no money in it.
16      */
17     public CashRegister()
18     {
19         purchase = 0;
20         payment = 0;

```

```

21 }
22
23 /**
24  Records the purchase price of an item.
25  @param amount the price of the purchased item
26  */
27 public void recordPurchase(double amount)
28 {
29     purchase = purchase + amount;
30 }
31
32 /**
33  Processes the payment received from the customer.
34  @param dollars the number of dollars in the payment
35  @param quarters the number of quarters in the payment
36  @param dimes the number of dimes in the payment
37  @param nickels the number of nickels in the payment
38  @param pennies the number of pennies in the payment
39  */
40 public void receivePayment(int dollars, int quarters,
41     int dimes, int nickels, int pennies)
42 {
43     payment = dollars + quarters * QUARTER_VALUE + dimes * DIME_VALUE
44         + nickels * NICKEL_VALUE + pennies * PENNY_VALUE;
45 }
46
47 /**
48  Computes the change due and resets the machine for the next customer.
49  @return the change due to the customer
50  */
51 public double giveChange()
52 {
53     double change = payment - purchase;
54     purchase = 0;
55     payment = 0;
56     return change;
57 }
58 }

```

section_1/CashRegisterTester.java

```

1 /**
2  This class tests the CashRegister class.
3  */
4 public class CashRegisterTester
5 {
6     public static void main(String[] args)
7     {
8         CashRegister register = new CashRegister();
9
10        register.recordPurchase(0.75);
11        register.recordPurchase(1.50);
12        register.receivePayment(2, 0, 5, 0, 0);
13        System.out.print("Change: ");
14        System.out.println(register.giveChange());
15        System.out.println("Expected: 0.25");
16
17        register.recordPurchase(2.25);
18        register.recordPurchase(19.25);
19        register.receivePayment(23, 2, 0, 0, 0);

```



```

20     System.out.print("Change: ");
21     System.out.println(register.giveChange());
22     System.out.println("Expected: 2.0");
23 }
24 }

```

Program Run

```

Change: 0.25
Expected: 0.25
Change: 2.0
Expected: 2.0

```

SELF CHECK

- Which are the most commonly used number types in Java?
- Suppose you want to write a program that works with population data from various countries. Which Java data type should you use?
- Which of the following initializations are incorrect, and why?
 - `int dollars = 100.0;`
 - `double balance = 100;`
- What is the difference between the following two statements?


```
final double CM_PER_INCH = 2.54;
```

 and


```
public static final double CM_PER_INCH = 2.54;
```
- What is wrong with the following statement sequence?


```
double diameter = . . . ;
double circumference = 3.14 * diameter;
```

Practice It Now you can try these exercises at the end of the chapter: R4.1, R4.21, E4.20.

Special Topic 4.1**Big Numbers**

If you want to compute with really large numbers, you can use big number objects. Big number objects are objects of the `BigInteger` and `BigDecimal` classes in the `java.math` package. Unlike the number types such as `int` or `double`, big number objects have essentially no limits on their size and precision. However, computations with big number objects are much slower than those that involve number types. Perhaps more importantly, you can't use the familiar arithmetic operators such as `(+ - *)` with them. Instead, you have to use methods called `add`, `subtract`, and `multiply`. Here is an example of how to create a `BigInteger` object and how to call the `multiply` method:

```

BigInteger n = new BigInteger("1000000");
BigInteger r = n.multiply(n);
System.out.println(r); // Prints 1000000000000

```

The `BigDecimal` type carries out floating-point computations without roundoff errors. For example,

```

BigDecimal d = new BigDecimal("4.35");
BigDecimal e = new BigDecimal("100");
BigDecimal f = d.multiply(e);
System.out.println(f); // Prints 435.00

```


Programming Tip 4.1

**Do Not Use Magic Numbers**

A **magic number** is a numeric constant that appears in your code without explanation. For example, consider the following scary example that actually occurs in the Java library source:

```
h = 31 * h + ch;
```

Why 31? The number of days in January? One less than the number of bits in an integer? Actually, this code computes a “hash code” from a string—a number that is derived from the characters in such a way that different strings are likely to yield different hash codes. The value 31 turns out to scramble the character values nicely.

A better solution is to use a named constant:

```
final int HASH_MULTIPLIER = 31;
h = HASH_MULTIPLIER * h + ch;
```

You should never use magic numbers in your code. Any number that is not completely self-explanatory should be declared as a named constant. Even the most reasonable cosmic constant is going to change one day. You think there are 365 days in a year? Your customers on Mars are going to be pretty unhappy about your silly prejudice. Make a constant

```
final int DAYS_PER_YEAR = 365;
```



We prefer programs that are easy to understand over those that appear to work by magic.

4.2 Arithmetic

In this section, you will learn how to carry out arithmetic calculations in Java.

4.2.1 Arithmetic Operators



Java supports the same four basic arithmetic operations as a calculator—addition, subtraction, multiplication, and division—but it uses different symbols for the multiplication and division **operators**.

You must write $a * b$ to denote multiplication. Unlike in mathematics, you cannot write $a \cdot b$, $a \cdot b$, or $a \times b$. Similarly, division is always indicated with the $/$ operator, never $a \div$ or a fraction bar. For example, $\frac{a+b}{2}$ becomes $(a + b) / 2$.

The combination of variables, literals, operators, and/or method calls is called an **expression**. For example, $(a + b) / 2$ is an expression.

Parentheses are used just as in algebra: to indicate in which order the parts of the expression should be computed. For example, in the expression $(a + b) / 2$, the sum $a + b$ is computed first, and then the sum is divided by 2. In contrast, in the expression

$$a + b / 2$$

only b is divided by 2, and then the sum of a and $b / 2$ is formed. As in regular algebraic notation, multiplication and division have a *higher precedence* than addition and subtraction. For example, in the expression $a + b / 2$, the $/$ is carried out first, even though the $+$ operation occurs further to the left (see Appendix B).

If you mix integer and floating-point values in an arithmetic expression, the result is a floating-point value. For example, $7 + 4.0$ is the floating-point value 11.0.

Mixing integers and floating-point values in an arithmetic expression yields a floating-point value.

4.2.2 Increment and Decrement

The ++ operator adds 1 to a variable; the -- operator subtracts 1.

Changing a variable by adding or subtracting 1 is so common that there is a special shorthand for it. The ++ operator increments a variable (see Figure 1):

```
counter++; // Adds 1 to the variable counter
```

Similarly, the -- operator decrements a variable:

```
counter--; // Subtracts 1 from counter
```

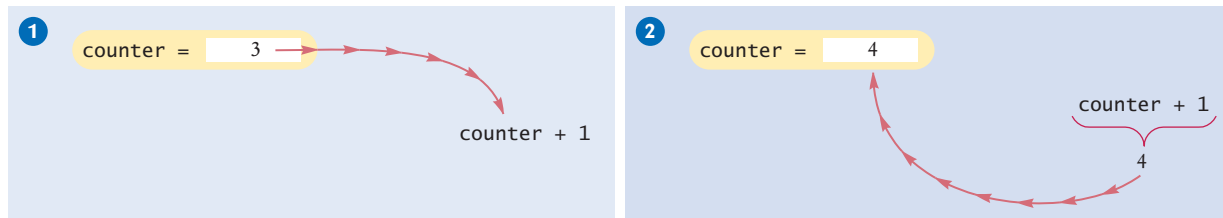


Figure 1 Incrementing a Variable

4.2.3 Integer Division and Remainder

If both arguments of / are integers, the remainder is discarded.

Division works as you would expect, as long as at least one of the numbers involved is a floating-point number. That is,

```
7.0 / 4.0
7 / 4.0
7.0 / 4
```

all yield 1.75. However, if *both* numbers are integers, then the result of the **integer division** is always an integer, with the remainder discarded. That is,

```
7 / 4
```

evaluates to 1 because 7 divided by 4 is 1 with a remainder of 3 (which is discarded). This can be a source of subtle programming errors—see Common Error 4.1.

If you are interested in the remainder only, use the % operator:

```
7 % 4
```

is 3, the remainder of the integer division of 7 by 4. The % symbol has no analog in algebra. It was chosen because it looks similar to /, and the remainder operation is related to division. The operator is called **modulus**. (Some people call it *modulo* or *mod*.) It has no relationship with the percent operation that you find on some calculators.

Here is a typical use for the integer / and % operations. Suppose you have an amount of pennies in a piggybank:

```
int pennies = 1729;
```

You want to determine the value in dollars and cents. You obtain the dollars through an integer division by 100:

```
int dollars = pennies / 100; // Sets dollars to 17
```



Integer division and the % operator yield the dollar and cent values of a piggybank full of pennies.

The % operator computes the remainder of an integer division.

Table 3 Integer Division and Remainder

Expression (where n = 1729)	Value	Comment
n % 10	9	n % 10 is always the last digit of n.
n / 10	172	This is always n without the last digit.
n % 100	29	The last two digits of n.
n / 10.0	172.9	Because 10.0 is a floating-point number, the fractional part is not discarded.
-n % 10	-9	Because the first argument is negative, the remainder is also negative.
n % 2	1	n % 2 is 0 if n is even, 1 or -1 if n is odd.

The integer division discards the remainder. To obtain the remainder, use the % operator:

```
int cents = pennies % 100; // Sets cents to 29
```

See Table 3 for additional examples.

4.2.4 Powers and Roots

The Java library declares many mathematical functions, such as `Math.sqrt` (square root) and `Math.pow` (raising to a power).

In Java, there are no symbols for powers and roots. To compute them, you must call methods. To take the square root of a number, you use the `Math.sqrt` method. For example, \sqrt{x} is written as `Math.sqrt(x)`. To compute x^n , you write `Math.pow(x, n)`.

In algebra, you use fractions, exponents, and roots to arrange expressions in a compact two-dimensional form. In Java, you have to write all expressions in a linear arrangement. For example, the mathematical expression

$$b \times \left(1 + \frac{r}{100}\right)^n$$

becomes

```
b * Math.pow(1 + r / 100, n)
```

Figure 2 shows how to analyze such an expression. Table 4 shows additional mathematical methods.

$$\begin{array}{c}
 b * \text{Math.pow}(1 + r / 100, n) \\
 \underbrace{\hspace{1.5cm}}_{\frac{r}{100}} \\
 \underbrace{\hspace{1.5cm}}_{1 + \frac{r}{100}} \\
 \underbrace{\hspace{1.5cm}}_{\left(1 + \frac{r}{100}\right)^n} \\
 \underbrace{\hspace{1.5cm}}_{b \times \left(1 + \frac{r}{100}\right)^n}
 \end{array}$$

Figure 2
Analyzing an Expression

Table 4 Mathematical Methods

Method	Returns	Method	Returns
<code>Math.sqrt(x)</code>	Square root of x (≥ 0)	<code>Math.abs(x)</code>	Absolute value $ x $
<code>Math.pow(x, y)</code>	x^y ($x > 0$, or $x = 0$ and $y > 0$, or $x < 0$ and y is an integer)	<code>Math.max(x, y)</code>	The larger of x and y
<code>Math.sin(x)</code>	Sine of x (x in radians)	<code>Math.min(x, y)</code>	The smaller of x and y
<code>Math.cos(x)</code>	Cosine of x	<code>Math.exp(x)</code>	e^x
<code>Math.tan(x)</code>	Tangent of x	<code>Math.log(x)</code>	Natural log ($\ln(x)$, $x > 0$)
<code>Math.round(x)</code>	Closest integer to x (as a <code>long</code>)	<code>Math.log10(x)</code>	Decimal log ($\log_{10}(x)$, $x > 0$)
<code>Math.ceil(x)</code>	Smallest integer $\geq x$ (as a <code>double</code>)	<code>Math.floor(x)</code>	Largest integer $\leq x$ (as a <code>double</code>)
<code>Math.toRadians(x)</code>	Convert x degrees to radians (i.e., returns $x \cdot \pi/180$)	<code>Math.toDegrees(x)</code>	Convert x radians to degrees (i.e., returns $x \cdot 180/\pi$)

4.2.5 Converting Floating-Point Numbers to Integers

Occasionally, you have a value of type `double` that you need to convert to the type `int`. It is an error to assign a floating-point value to an integer:

```
double balance = total + tax;
int dollars = balance; // Error: Cannot assign double to int
```

The compiler disallows this assignment because it is potentially dangerous:

- The fractional part is lost.
- The magnitude may be too large. (The largest integer is about 2 billion, but a floating-point number can be much larger.)

You use a cast (*typeName*) to convert a value to a different type.

You must use the **cast** operator (`int`) to convert a floating-point value to an integer. Write the cast operator before the expression that you want to convert:

```
double balance = total + tax;
int dollars = (int) balance;
```

The cast (`int`) converts the floating-point value `balance` to an integer by discarding the fractional part. For example, if `balance` is 13.75, then `dollars` is set to 13.

When applying the cast operator to an arithmetic expression, you need to place the expression inside parentheses:

```
int dollars = (int) (total + tax);
```

Discarding the fractional part is not always appropriate. If you want to round a floating-point number to the nearest whole number, use the `Math.round` method. This method returns a `long` integer, because large floating-point numbers cannot be stored in an `int`.

```
long rounded = Math.round(balance);
```

If `balance` is 13.75, then `rounded` is set to 14.

FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download a program that demonstrates casts, rounding, and the `%` operator.

Syntax 4.2 Cast

Syntax

(typeName)

expression

This is the type of the expression after casting.

These parentheses are a part of the cast operator.

(int)

(balance * 100)

Use parentheses here if the cast is applied to an expression with arithmetic operators.

If you know that the result can be stored in an `int` and does not require a `long`, you can use a cast:

```
int rounded = (int) Math.round(balance);
```

Table 5 Arithmetic Expressions		
Mathematical Expression	Java Expression	Comments
$\frac{x + y}{2}$	<code>(x + y) / 2</code>	The parentheses are required; <code>x + y / 2</code> computes $x + \frac{y}{2}$.
$\frac{xy}{2}$	<code>x * y / 2</code>	Parentheses are not required; operators with the same precedence are evaluated left to right.
$\left(1 + \frac{r}{100}\right)^n$	<code>Math.pow(1 + r / 100, n)</code>	Use <code>Math.pow(x, n)</code> to compute x^n .
$\sqrt{a^2 + b^2}$	<code>Math.sqrt(a * a + b * b)</code>	<code>a * a</code> is simpler than <code>Math.pow(a, 2)</code> .
$\frac{i + j + k}{3}$	<code>(i + j + k) / 3.0</code>	If <i>i</i> , <i>j</i> , and <i>k</i> are integers, using a denominator of 3.0 forces floating-point division.
π	<code>Math.PI</code>	<code>Math.PI</code> is a constant declared in the <code>Math</code> class.



- 6. A bank account earns interest once per year. In Java, how do you compute the interest earned in the first year? Assume variables `percent` and `balance` of type `double` have already been declared.
- 7. In Java, how do you compute the side length of a square whose area is stored in the variable `area`?
- 8. The volume of a sphere is given by

$$V = \frac{4}{3}\pi r^3$$

If the radius is given by a variable `radius` of type `double`, write a Java expression for the volume.

9. What is the value of $1729 / 100$ and $1729 \% 100$?
 10. If n is a positive number, what is $(n / 10) \% 10$?

Practice It Now you can try these exercises at the end of the chapter: R4.4, R4.6, E4.4, E4.23.

Common Error 4.1



Unintended Integer Division

It is unfortunate that Java uses the same symbol, namely `/`, for both integer and floating-point division. These are really quite different operations. It is a common error to use **integer division** by accident. Consider this segment that computes the average of three integers:

```
int score1 = 10;
int score2 = 4;
int score3 = 9;

double average = (score1 + score2 + score3) / 3; // Error
System.out.println("Average score: " + average); // Prints 7.0, not 7.666666666666667
```

What could be wrong with that? Of course, the average of `score1`, `score2`, and `score3` is

$$\frac{\text{score1} + \text{score2} + \text{score3}}{3}$$

Here, however, the `/` does not mean division in the mathematical sense. It denotes integer division because both 3 and the sum of `score1 + score2 + score3` are integers. Because the scores add up to 23, the average is computed to be 7, the result of the integer division of 23 by 3. That integer 7 is then moved into the floating-point variable `average`. The remedy is to make the numerator or denominator into a floating-point number:

```
double total = score1 + score2 + score3;
double average = total / 3;

or

double average = (score1 + score2 + score3) / 3.0;
```

Common Error 4.2



Unbalanced Parentheses

Consider the expression

$$((a + b) * t / 2 * (1 - t))$$

What is wrong with it? Count the parentheses. There are three `(` and two `)`. The parentheses are *unbalanced*. This kind of typing error is very common with complicated expressions. Now consider this expression.

$$(a + b) * t) / (2 * (1 - t))$$

This expression has three `(` and three `)`, but it still is not correct. In the middle of the expression,

$$(a + b) * t) / (2 * (1 - t))$$

↑

there is only one `(` but two `)`, which is an error. In the middle of an expression, the count of `(` must be greater than or equal to the count of `)`, and at the end of the expression the two counts must be the same.

Here is a simple trick to make the counting easier without using pencil and paper. It is difficult for the brain to keep two counts simultaneously. Keep only one count when scanning the expression. Start with 1 at the first opening parenthesis, add 1 whenever



you see an opening parenthesis, and subtract one whenever you see a closing parenthesis. Say the numbers aloud as you scan the expression. If the count ever drops below zero, or is not zero at the end, the parentheses are unbalanced. For example, when scanning the previous expression, you would mutter

```
(a + b) * t) / (2 * (1 - t)
1   0  -1
```

and you would find the error.

Programming Tip 4.2



Spaces in Expressions

It is easier to read

```
x1 = (-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a);
```

than

```
x1=(-b+Math.sqrt(b*b-4*a*c))/(2*a);
```

Simply put spaces around all operators `+` `-` `*` `/` `%` `=`. However, don't put a space after a *unary* minus: `a -` used to negate a single quantity, such as `-b`. That way, it can be easily distinguished from a *binary* minus, as in `a - b`.

It is customary not to put a space after a method name. That is, write `Math.sqrt(x)` and not `Math.sqrt (x)`.

Special Topic 4.2



Combining Assignment and Arithmetic

In Java, you can combine arithmetic and assignment. For example, the instruction

```
balance += amount;
```

is a shortcut for

```
balance = balance + amount;
```

Similarly,

```
total *= 2;
```

is another way of writing

```
total = total * 2;
```

Many programmers find this a convenient shortcut. If you like it, go ahead and use it in your own code. For simplicity, we won't use it in this book, though.

Special Topic 4.3



Instance Methods and Static Methods

In the preceding section, you encountered the `Math` class, which contains a collection of helpful methods for carrying out mathematical computations. These methods do not operate on an object. That is, you don't call

```
double root = 2.sqrt(); // Error
```

In Java, numbers are not objects, so you can never invoke a method on a number. Instead, you pass a number as an argument (explicit parameter) to a method, enclosing the number in parentheses after the method name:

```
double root = Math.sqrt(2);
```


Such methods are called **static methods**. (The term “static” is a historical holdover from the C and C++ programming languages. It has nothing to do with the usual meaning of the word.)

Static methods do not operate on objects, but they are still declared inside classes. When calling the method, you specify the class to which the `sqrt` method belongs:

The name of the class
The name of the static method
`Math.sqrt(2)`

In contrast, a method that is invoked on an object is called an **instance method**. As a rule of thumb, you use static methods when you manipulate numbers. You will learn more about the distinction between static and instance methods in Chapter 8.



Computing & Society 4.1 The Pentium Floating-Point Bug

In 1994, Intel Corporation released what was then its most powerful processor, the Pentium. Unlike previous generations of its processors, it had a very fast floating-point unit. Intel's goal was to compete aggressively with the makers of higher-end processors for engineering workstations. The Pentium was a huge success immediately.

In the summer of 1994, Dr. Thomas Nicely of Lynchburg College in Virginia ran an extensive set of computations to analyze the sums of reciprocals of certain sequences of prime numbers. The results were not always what his theory predicted, even after he took into account the inevitable roundoff errors. Then Dr. Nicely noted that the same program did produce the correct results when running on the slower 486 processor that preceded the Pentium in Intel's lineup. This should not have happened. The optimal round-off behavior of floating-point calculations has been standardized by the Institute for Electrical and Electronic Engineers (IEEE) and Intel claimed to adhere to the IEEE standard in both the 486 and the Pentium processors. Upon further checking, Dr. Nicely discovered that indeed there was a very small set of numbers for which the product of two numbers was computed differently on the two processors. For example,

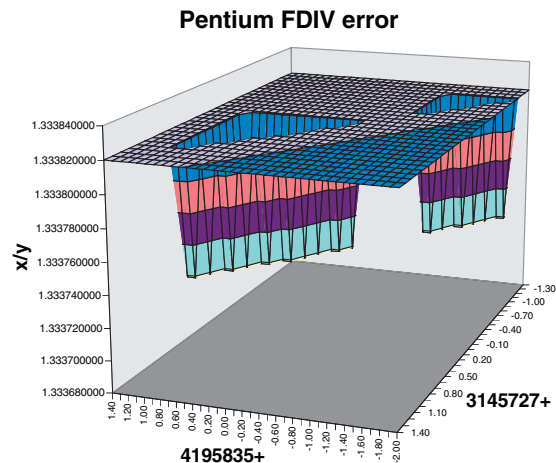
$$4,195,835 - ((4,195,835/3,145,727) \times 3,145,727)$$

is mathematically equal to 0, and it did compute as 0 on a 486 processor. On his Pentium processor the result was 256.

As it turned out, Intel had independently discovered the bug in its testing and had started to produce chips that fixed it. The bug was caused by an error in a table that was used to speed up the floating-point multiplication algorithm of the processor. Intel determined that the problem was exceedingly rare. They claimed that under normal use, a typical consumer would only notice the problem once every 27,000 years. Unfortunately for Intel, Dr. Nicely had not been a normal user.

Now Intel had a real problem on its hands. It figured that the cost of replacing all Pentium processors that it had sold so far would cost a great deal of money. Intel already had more orders for the chip than it could produce, and it would be particularly galling to have to give out the scarce chips as free replacements instead of selling them. Intel's management decided to punt on the issue and initially offered to replace the processors only for those customers who could prove that their work required absolute precision in mathematical calculations. Naturally, that did not go over well with the hundreds of thousands of customers who had paid retail prices of \$700 and more for a Pentium chip and did not want to live with the nagging feeling that perhaps, one day, their income tax program would produce a faulty return.

Ultimately, Intel caved in to public demand and replaced all defective chips, at a cost of about 475 million dollars.



This graph shows a set of numbers for which the original Pentium processor obtained the wrong quotient.

4.3 Input and Output

In the following sections, you will see how to read user input and how to control the appearance of the output that your programs produce.

4.3.1 Reading Input



A supermarket scanner reads bar codes. The Java Scanner reads numbers and text.

You can make your programs more flexible if you ask the program user for inputs rather than using fixed values. Consider, for example, a program that processes prices and quantities of soda containers. Prices and quantities are likely to fluctuate. The program user should provide them as inputs.

When a program asks for user input, it should first print a message that tells the user which input is expected. Such a message is called a **prompt**.

```
System.out.print("Please enter the number of bottles: "); // Display prompt
```

Use the `print` method, not `println`, to display the prompt. You want the input to appear after the colon, not on the following line. Also remember to leave a space after the colon.

Because output is sent to `System.out`, you might think that you use `System.in` for input. Unfortunately, it isn't quite that simple. When Java was first designed, not much attention was given to reading keyboard input. It was assumed that all programmers would produce graphical user interfaces with text fields and menus. `System.in` was given a minimal set of features and must be combined with other classes to be useful.

To read keyboard input, you use a class called `Scanner`. You obtain a `Scanner` object by using the following statement:

```
Scanner in = new Scanner(System.in);
```

Once you have a scanner, you use its `nextInt` method to read an integer value:

```
System.out.print("Please enter the number of bottles: ");
int bottles = in.nextInt();
```

Use the `Scanner` class to read keyboard input in a console window.

Syntax 4.3 Input Statement

```

Include this line so you can use the Scanner class.  import java.util.Scanner;
.
.
Create a Scanner object to read keyboard input.  Scanner in = new Scanner(System.in);
.
.
Display a prompt in the console window.  System.out.print("Please enter the number of bottles: ");
.
Define a variable to hold the input value.  int bottles = in.nextInt();

Don't use println here.

The program waits for user input, then places the input into the variable.

```

When the `nextInt` method is called, the program waits until the user types a number and presses the Enter key. After the user supplies the input, the number is placed into the `bottles` variable, and the program continues.

To read a floating-point number, use the `nextDouble` method instead:

```
System.out.print("Enter price: ");
double price = in.nextDouble();
```

The `Scanner` class belongs to the package `java.util`. When using the `Scanner` class, import it by placing the following declaration at the top of your program file:

```
import java.util.Scanner;
```

4.3.2 Formatted Output

When you print the result of a computation, you often want to control its appearance. For example, when you print an amount in dollars and cents, you usually want it to be rounded to two significant digits. That is, you want the output to look like

Price per liter: 1.22

instead of

Price per liter: 1.215962441314554

The following command displays the price with two digits after the decimal point:

```
System.out.printf("%.2f", price);
```

You can also specify a *field width*:

```
System.out.printf("%10.2f", price);
```

The price is printed using ten characters: six spaces followed by the four characters 1.22.

```
      1 . 2 2
```

The construct `%10.2f` is called a *format specifier*: it describes how a value should be formatted. The letter `f` at the end of the format specifier indicates that we are displaying a floating-point number. Use `d` for an integer and `s` for a string; see Table 6 for examples. A format string contains format specifiers and literal characters. Any characters that are not format specifiers are printed verbatim. For example, the command

```
System.out.printf("Price per liter:%10.2f", price);
```

prints

Price per liter: 1.22

Use the `printf` method to specify how values should be formatted.

You use the `printf` method to line up your output in neat columns.

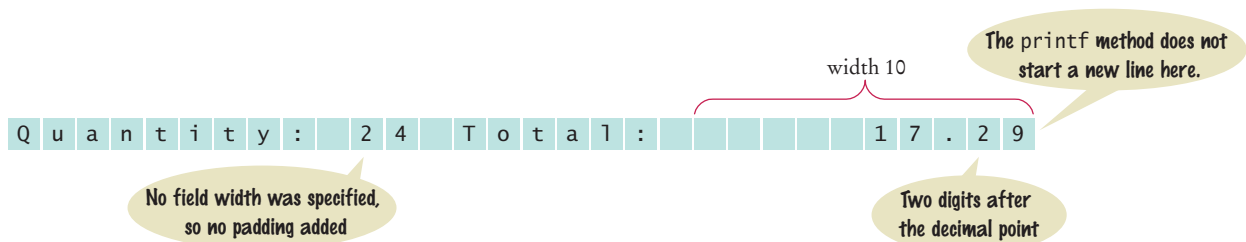
	Commencement			Term	Expirat	
	Month	Day	Year		Month	Day
Ornewaart	June	4	1920	5-yr	June	14
Ornewaart	April	24	1920	5-yr	April	24
slager	Mar	14	1923	5-yr	Mar	14
slager	Feb	9	1925	5-yr	Mar	14
slager	Oct	20	1925	5-yr	Oct	20
Mr. Massard Capt. B. B. B.	Mar	15	1924	5-yr	Mar	15
Quin	Nov.	14	1924	5-yr	Nov	14
Sr.	Sept	5	1925	5-yr	Sept	5
Dongle	May	22	1926	5-yr	May	22
I	Mar	Sh.	1925		Oct	25
liger	Mar	14	1928	5-yr	Mar	14

Table 6 Format Specifier Examples

Format String	Sample Output	Comments
"%d"	24	Use d with an integer.
"%5d"	24	Spaces are added so that the field width is 5.
"Quantity:%5d"	Quantity: 24	Characters inside a format string but outside a format specifier appear in the output.
"%f"	1.21997	Use f with a floating-point number.
"%.2f"	1.22	Prints two digits after the decimal point.
"%7.2f"	1.22	Spaces are added so that the field width is 7.
"%s"	Hello	Use s with a string.
"%d %.2f"	24 1.22	You can format multiple values at once.

You can print multiple values with a single call to the `printf` method. Here is a typical example:

```
System.out.printf("Quantity: %d Total: %10.2f", quantity, total);
```



The `printf` method, like the `print` method, does not start a new line after the output. If you want the next output to be on a separate line, you can call `System.out.println()`. Alternatively, Section 4.5.4 shows you how to add a newline character to the format string.

Our next example program will prompt for the price of a six-pack of soda and a two-liter bottle, and then print out the price per liter for both. The program puts to work what you just learned about reading input and formatting output.



What is the better deal? A six-pack of 12-ounce cans or a two-liter bottle?

section_3/Volume.java

```

1  import java.util.Scanner;
2
3  /**
4   * This program prints the price per liter for a six-pack of cans and
5   * a two-liter bottle.
6   */
7  public class Volume
8  {
9      public static void main(String[] args)
10     {
11         // Read price per pack
12
13         Scanner in = new Scanner(System.in);
14
15         System.out.print("Please enter the price for a six-pack: ");
16         double packPrice = in.nextDouble();
17
18         // Read price per bottle
19
20         System.out.print("Please enter the price for a two-liter bottle: ");
21         double bottlePrice = in.nextDouble();
22
23         final double CANS_PER_PACK = 6;
24         final double CAN_VOLUME = 0.355; // 12 oz. = 0.355 l
25         final double BOTTLE_VOLUME = 2;
26
27         // Compute and print price per liter
28
29         double packPricePerLiter = packPrice / (CANS_PER_PACK * CAN_VOLUME);
30         double bottlePricePerLiter = bottlePrice / BOTTLE_VOLUME;
31
32         System.out.printf("Pack price per liter: %8.2f", packPricePerLiter);
33         System.out.println();
34
35         System.out.printf("Bottle price per liter: %8.2f", bottlePricePerLiter);
36         System.out.println();
37     }
38 }

```

Program Run

```

Please enter the price for a six-pack: 2.95
Please enter the price for a two-liter bottle: 2.85
Pack price per liter:      1.38
Bottle price per liter:    1.43

```



11. Write statements to prompt for and read the user's age using a Scanner variable named in.
12. What is wrong with the following statement sequence?

```

System.out.print("Please enter the unit price: ");
double unitPrice = in.nextDouble();
int quantity = in.nextInt();

```

13. What is problematic about the following statement sequence?

```
System.out.print("Please enter the unit price: ");
double unitPrice = in.nextInt();
```

14. What is problematic about the following statement sequence?

```
System.out.print("Please enter the number of cans");
int cans = in.nextInt();
```

15. What is the output of the following statement sequence?

```
int volume = 10;
System.out.printf("The volume is %5d", volume);
```

16. Using the printf method, print the values of the integer variables bottles and cans so that the output looks like this:

```
Bottles:      8
Cans:         24
```

The numbers to the right should line up. (You may assume that the numbers have at most 8 digits.)

Practice It Now you can try these exercises at the end of the chapter: R4.11, E4.6, E4.7.

HOW TO 4.1

Carrying Out Computations



Many programming problems require arithmetic computations. This How To shows you how to turn a problem statement into pseudocode and, ultimately, a Java program.

Problem Statement Suppose you are asked to write a program that simulates a vending machine. A customer selects an item for purchase and inserts a bill into the vending machine. The vending machine dispenses the purchased item and gives change. We will assume that all item prices are multiples of 25 cents, and the machine gives all change in dollar coins and quarters. Your task is to compute how many coins of each type to return.

- Step 1** Understand the problem: What are the inputs? What are the desired outputs?

In this problem, there are two inputs:

- The denomination of the bill that the customer inserts
- The price of the purchased item

There are two desired outputs:

- The number of dollar coins that the machine returns
- The number of quarters that the machine returns

- Step 2** Work out examples by hand.

This is a very important step. If you can't compute a couple of solutions by hand, it's unlikely that you'll be able to write a program that automates the computation.

Let's assume that a customer purchased an item that cost \$2.25 and inserted a \$5 bill. The customer is due \$2.75, or two dollar coins and three quarters, in change.

That is easy for you to see, but how can a Java program come to the same conclusion? The key is to work in pennies, not dollars. The change due the customer is 275 pennies. Dividing by 100 yields 2, the number of dollars. Dividing the remainder (75) by 25 yields 3, the number of quarters.

Step 3 Write pseudocode for computing the answers.

In the previous step, you worked out a specific instance of the problem. You now need to come up with a method that works in general.

Given an arbitrary item price and payment, how can you compute the coins due? First, compute the change due in pennies:

change due = 100 x bill value - item price in pennies

To get the dollars, divide by 100 and discard the remainder:

dollar coins = change due / 100 (without remainder)

The remaining change due can be computed in two ways. If you are familiar with the modulus operator, you can simply compute

change due = change due % 100

Alternatively, subtract the penny value of the dollar coins from the change due:

change due = change due - 100 x dollar coins

To get the quarters due, divide by 25:

quarters = change due / 25

Step 4 Declare the variables and constants that you need, and specify their types.

Here, we have five variables:

- billValue
- itemPrice
- changeDue
- dollarCoins
- quarters

Should we introduce constants to explain 100 and 25 as `PENNIES_PER_DOLLAR` and `PENNIES_PER_QUARTER`? Doing so will make it easier to convert the program to international markets, so we will take this step.

It is very important that `changeDue` and `PENNIES_PER_DOLLAR` are of type `int` because the computation of `dollarCoins` uses integer division. Similarly, the other variables are integers.

Step 5 Turn the pseudocode into Java statements.

If you did a thorough job with the pseudocode, this step should be easy. Of course, you have to know how to express mathematical operations (such as powers or integer division) in Java.

```
changeDue = PENNIES_PER_DOLLAR * billValue - itemPrice;
dollarCoins = changeDue / PENNIES_PER_DOLLAR;
changeDue = changeDue % PENNIES_PER_DOLLAR;
quarters = changeDue / PENNIES_PER_QUARTER;
```

Step 6 Provide input and output.

Before starting the computation, we prompt the user for the bill value and item price:

```
System.out.print("Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ");
billValue = in.nextInt();
System.out.print("Enter item price in pennies: ");
itemPrice = in.nextInt();
```

When the computation is finished, we display the result. For extra credit, we use the `printf` method to make sure that the output lines up neatly.

```
System.out.printf("Dollar coins: %6d", dollarCoins);
System.out.printf("Quarters:    %6d", quarters);
```


A vending machine takes bills and gives change in coins.



Step 7 Provide a class with a main method.

Your computation needs to be placed into a class. Find an appropriate name for the class that describes the purpose of the computation. In our example, we will choose the name `VendingMachine`.

Inside the class, supply a main method.

In the main method, you need to declare constants and variables (Step 4), carry out computations (Step 5), and provide input and output (Step 6). Clearly, you will want to first get the input, then do the computations, and finally show the output. Declare the constants at the beginning of the method, and declare each variable just before it is needed.

Here is the complete program, `how_to_1/VendingMachine.java`:

```
import java.util.Scanner;

/**
 * This program simulates a vending machine that gives change.
 */
public class VendingMachine
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        final int PENNIES_PER_DOLLAR = 100;
        final int PENNIES_PER_QUARTER = 25;

        System.out.print("Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ");
        int billValue = in.nextInt();
        System.out.print("Enter item price in pennies: ");
        int itemPrice = in.nextInt();

        // Compute change due

        int changeDue = PENNIES_PER_DOLLAR * billValue - itemPrice;
        int dollarCoins = changeDue / PENNIES_PER_DOLLAR;
        changeDue = changeDue % PENNIES_PER_DOLLAR;
        int quarters = changeDue / PENNIES_PER_QUARTER;

        // Print change due

        System.out.printf("Dollar coins: %d", dollarCoins);
        System.out.println();
    }
}
```

```

        System.out.printf("Quarters:    %6d", quarters);
        System.out.println();
    }
}

```

Program Run

```

Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): 5
Enter item price in pennies: 225
Dollar coins:    2
Quarters:       3

```

**WORKED EXAMPLE 4.1****Computing the Volume and Surface Area of a Pyramid**

Learn how to design a class for computing the volume and surface area of a pyramid. Go to wiley.com/go/javaexamples and download Worked Example 4.1.



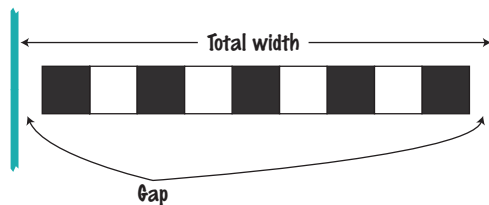
4.4 Problem Solving: First Do It By Hand

A very important step for developing an algorithm is to first carry out the computations *by hand*. If you can't compute a solution yourself, it's unlikely that you'll be able to write a program that automates the computation.

To illustrate the use of hand calculations, consider the following problem.

A row of black and white tiles needs to be placed along a wall. For aesthetic reasons, the architect has specified that the first and last tile shall be black.

Your task is to compute the number of tiles needed and the gap at each end, given the space available and the width of each tile.



Pick concrete values for a typical situation to use in a hand calculation.

To make the problem more concrete, let's assume the following dimensions:

- Total width: 100 inches
- Tile width: 5 inches

The obvious solution would be to fill the space with 20 tiles, but that would not work—the last tile would be white.

Instead, look at the problem this way: The first tile must always be black, and then we add some number of white/black pairs:



The first tile takes up 5 inches, leaving 95 inches to be covered by pairs. Each pair is 10 inches wide. Therefore the number of pairs is $95 / 10 = 9.5$. However, we need to discard the fractional part since we can't have fractions of tile pairs.

Therefore, we will use 9 tile pairs or 18 tiles, plus the initial black tile. Altogether, we require 19 tiles.

The tiles span $19 \times 5 = 95$ inches, leaving a total gap of $100 - 19 \times 5 = 5$ inches.

The gap should be evenly distributed at both ends. At each end, the gap is $(100 - 19 \times 5) / 2 = 2.5$ inches.

This computation gives us enough information to devise an algorithm with arbitrary values for the total width and tile width.

number of pairs = integer part of $(\text{total width} - \text{tile width}) / (2 \times \text{tile width})$

number of tiles = $1 + 2 \times \text{number of pairs}$

gap at each end = $(\text{total width} - \text{number of tiles} \times \text{tile width}) / 2$

As you can see, doing a hand calculation gives enough insight into the problem that it becomes easy to develop an algorithm.

SELF CHECK



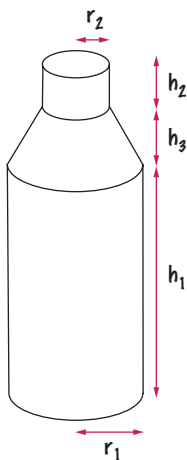
17. Translate the pseudocode for computing the number of tiles and the gap width into Java.
18. Suppose the architect specifies a pattern with black, gray, and white tiles, like this:



Again, the first and last tile should be black. How do you need to modify the algorithm?

19. A robot needs to tile a floor with alternating black and white tiles. Develop an algorithm that yields the color (0 for black, 1 for white), given the row and column number. Start with specific values for the row and column, and then generalize.

	1	2	3	4
1	Black	White	Black	White
2	White	Black	White	Black
3	Black	White	Black	White
4	White	Black	White	Black



20. For a particular car, repair and maintenance costs in year 1 are estimated at \$100; in year 10, at \$1,500. Assuming that the repair cost increases by the same amount every year, develop pseudocode to compute the repair cost in year 3 and then generalize to year n .
21. The shape of a bottle is approximated by two cylinders of radius r_1 and r_2 and heights h_1 and h_2 , joined by a cone section of height h_3 .

Using the formulas for the volume of a cylinder, $V = \pi r^2 h$, and a cone section,

$$V = \pi \frac{(r_1^2 + r_1 r_2 + r_2^2)h}{3},$$

develop pseudocode to compute the volume of the bottle. Using an actual bottle with known volume as a sample, make a hand calculation of your pseudocode.

Practice It Now you can try these exercises at the end of the chapter: R4.16, R4.18, R4.19.



WORKED EXAMPLE 4.2

Computing Travel Time

Learn how to develop a hand calculation to compute the time that a robot requires to retrieve an item from rocky terrain. Go to wiley.com/go/javaexamples and download Worked Example 4.2.



4.5 Strings

Strings are sequences of characters.

Many programs process text, not numbers. Text consists of **characters**: letters, numbers, punctuation, spaces, and so on. A **string** is a sequence of characters. For example, the string "Harry" is a sequence of five characters.



4.5.1 The String Type

You can declare variables that hold strings.

```
String name = "Harry";
```

We distinguish between string variables (such as the variable name declared above) and string **literals** (character sequences enclosed in quotes, such as "Harry"). A string variable is simply a variable that can hold a string, just as an integer variable can hold an integer. A string literal denotes a particular string, just as a number literal (such as 2) denotes a particular number.

The number of characters in a string is called the *length* of the string. For example, the length of "Harry" is 5. As you saw in Section 2.3, you can compute the length of a string with the `length` method.

```
int n = name.length();
```

A string of length 0 is called the *empty string*. It contains no characters and is written as "".

The `length` method yields the number of characters in a string.

4.5.2 Concatenation

Use the + operator to *concatenate* strings; that is, to put them together to yield a longer string.

Given two strings, such as "Harry" and "Morgan", you can **concatenate** them to one long string. The result consists of all characters in the first string, followed by all characters in the second string. In Java, you use the + operator to concatenate two strings.

For example,

```
String fName = "Harry";
String lName = "Morgan";
String name = fName + lName;
```

results in the string

"HarryMorgan"

What if you'd like the first and last name separated by a space? No problem:

```
String name = fName + " " + lName;
```

This statement concatenates three strings: `fName`, the string literal " ", and `lName`. The result is

"Harry Morgan"

When the expression to the left or the right of a + operator is a string, the other one is automatically forced to become a string as well, and both strings are concatenated.

For example, consider this code:

```
String jobTitle = "Agent";
int employeeId = 7;
String bond = jobTitle + employeeId;
```

Because `jobTitle` is a string, `employeeId` is converted from the integer 7 to the string "7". Then the two strings "Agent" and "7" are concatenated to form the string "Agent7".

This concatenation is very useful for reducing the number of `System.out.print` instructions. For example, you can combine

```
System.out.print("The total is ");
System.out.println(total);
```

to the single call

```
System.out.println("The total is " + total);
```

The concatenation "The total is " + `total` computes a single string that consists of the string "The total is ", followed by the string equivalent of the number `total`.

Whenever one of the arguments of the + operator is a string, the other argument is converted to a string.

4.5.3 String Input

Use the next method of the Scanner class to read a string containing a single word.

You can read a string from the console:

```
System.out.print("Please enter your name: ");
String name = in.next();
```

When a string is read with the next method, only one word is read. For example, suppose the user types

Harry Morgan

as the response to the prompt. This input consists of two words. The call `in.next()` yields the string "Harry". You can use another call to `in.next()` to read the second word.

4.5.4 Escape Sequences

To include a quotation mark in a literal string, precede it with a backslash (\), like this:

```
"He said \"Hello\""
```

The backslash is not included in the string. It indicates that the quotation mark that follows should be a part of the string and not mark the end of the string. The sequence \" is called an **escape sequence**.

To include a backslash in a string, use the escape sequence \\, like this:

```
"C:\\Temp\\Secret.txt"
```

Another common escape sequence is \n, which denotes a **newline** character. Printing a newline character causes the start of a new line on the display. For example, the statement

```
System.out.print("*\n**\n***\n");
```

prints the characters

```
*
**
***
```

on three separate lines.

You often want to add a newline character to the end of the format string when you use System.out.printf:

```
System.out.printf("Price: %10.2f\n", price);
```

4.5.5 Strings and Characters

Strings are sequences of **Unicode** characters (see Computing & Society 4.2). In Java, a **character** is a value of the type char. Characters have numeric values. You can find the values of the characters that are used in Western European languages in Appendix A. For example, if you look up the value for the character 'H', you can see that it is actually encoded as the number 72.

Character literals are delimited by single quotes, and you should not confuse them with strings.

- 'H' is a character, a value of type char.
- "H" is a string containing a single character, a value of type String.

The charAt method returns a char value from a string. The first string position is labeled 0, the second one 1, and so on.

H	a	r	r	y
0	1	2	3	4

The position number of the last character (4 for the string "Harry") is always one less than the length of the string.

String positions are counted starting with 0.



A string is a sequence of characters.

For example, the statement

```
String name = "Harry";
char start = name.charAt(0);
char last = name.charAt(4);
```

sets `start` to the value 'H' and `last` to the value 'y'.

4.5.6 Substrings

Use the `substring` method to extract a part of a string.

Once you have a string, you can extract substrings by using the `substring` method. The method call

```
str.substring(start, pastEnd)
```

returns a string that is made up of the characters in the string `str`, starting at position `start`, and containing all characters up to, but not including, the position `pastEnd`. Here is an example:

```
String greeting = "Hello, World!";
String sub = greeting.substring(0, 5); // sub is "Hello"
```

Here the `substring` operation makes a string that consists of the first five characters taken from the string `greeting`.

H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

Let's figure out how to extract the substring "World". Count characters starting at 0, not 1. You find that `W` has position number 7. The first character that you don't want, `!`, is the character at position 12. Therefore, the appropriate substring command is

```
String sub2 = greeting.substring(7, 12);
```

H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

5

↑

It is curious that you must specify the position of the first character that you do want and then the first character that you don't want. There is one advantage to this setup. You can easily compute the length of the substring: It is `pastEnd - start`. For example, the string "World" has length $12 - 7 = 5$.

If you omit the end position when calling the `substring` method, then all characters from the starting position to the end of the string are copied. For example,

```
String tail = greeting.substring(7); // Copies all characters from position 7 on
```

sets `tail` to the string "World!".

Following is a simple program that puts these concepts to work. The program asks for your name and that of your significant other. It then prints out your initials.

The operation `first.substring(0, 1)` makes a string consisting of one character, taken from the start of `first`. The program does the same for the second. Then it concatenates the resulting one-character strings with the string literal `"&"` to get a string of length 3, the initials string. (See Figure 3.)

first =	R	o	d	o	l	f	o
	0	1	2	3	4	5	6
second =	S	a	l	l	y		
	0	1	2	3	4		
initials =	R	&	S				
	0	1	2				

Figure 3 Building the initials String



Initials are formed from the first letter of each name.

section_5/Initials.java

```

1  import java.util.Scanner;
2
3  /**
4   * This program prints a pair of initials.
5   */
6  public class Initials
7  {
8      public static void main(String[] args)
9      {
10         Scanner in = new Scanner(System.in);
11
12         // Get the names of the couple
13
14         System.out.print("Enter your first name: ");
15         String first = in.next();
16         System.out.print("Enter your significant other's first name: ");
17         String second = in.next();
18
19         // Compute and display the inscription
20
21         String initials = first.substring(0, 1)
22             + "&" + second.substring(0, 1);
23         System.out.println(initials);
24     }
25 }

```

Program Run

```

Enter your first name: Rodolfo
Enter your significant other's first name: Sally
R&S

```

Table 7 String Operations

Statement	Result	Comment
<code>string str = "Ja"; str = str + "va";</code>	str is set to "Java"	When applied to strings, + denotes concatenation.
<code>System.out.println("Please" + " enter your name: ");</code>	Prints Please enter your name:	Use concatenation to break up strings that don't fit into one line.
<code>team = 49 + "ers"</code>	team is set to "49ers"	Because "ers" is a string, 49 is converted to a string.
<code>String first = in.next(); String last = in.next(); (User input: Harry Morgan)</code>	first contains "Harry" last contains "Morgan"	The next method places the next word into the string variable.
<code>String greeting = "H & S"; int n = greeting.length();</code>	n is set to 5	Each space counts as one character.
<code>String str = "Sally"; char ch = str.charAt(1);</code>	ch is set to 'a'	This is a char value, not a String. Note that the initial position is 0.
<code>String str = "Sally"; String str2 = str.substring(1, 4);</code>	str2 is set to "all"	Extracts the substring starting at position 1 and ending before position 4.
<code>String str = "Sally"; String str2 = str.substring(1);</code>	str2 is set to "ally"	If you omit the end position, all characters from the position until the end of the string are included.
<code>String str = "Sally"; String str2 = str.substring(1, 2);</code>	str2 is set to "a"	Extracts a String of length 1; contrast with <code>str.charAt(1)</code> .
<code>String last = str.substring(str.length() - 1);</code>	last is set to the string containing the last character in str	The last character has position <code>str.length() - 1</code> .



22. What is the length of the string "Java Program"?
23. Consider this string variable.

```
String str = "Java Program";
```

 Give a call to the substring method that returns the substring "gram".
24. Use string concatenation to turn the string variable str from Self Check 23 into "Java Programming".
25. What does the following statement sequence print?

```
String str = "Harry";  
int n = str.length();  
String mystery = str.substring(0, 1) + str.substring(n - 1, n);  
System.out.println(mystery);
```
26. Give an input statement to read a name of the form "John Q. Public".

Practice It Now you can try these exercises at the end of the chapter: R4.8, R4.12, E4.14, P4.6.

Programming Tip 4.3



Reading Exception Reports

You will often have programs that terminate and display an error message, such as

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
    String index out of range: -4
    at java.lang.String.substring(String.java:1444)
    at Homework1.main(Homework1.java:16)
```

If this happens to you, don't say "it didn't work," or "my program died." Instead, read the error message. Admittedly, the format of the exception report is not very friendly. But it is actually easy to decipher it.

When you have a close look at the error message, you will notice two pieces of useful information:

1. The name of the exception, such as `StringIndexOutOfBoundsException`
2. The line number of the code that contained the statement that caused the exception, such as `Homework1.java:16`

The name of the exception is always in the first line of the report, and it ends in `Exception`. If you get a `StringIndexOutOfBoundsException`, then there was a problem with accessing an invalid position in a string. That is useful information.

The line number of the offending code is a little harder to determine. The exception report contains the entire **stack trace**—that is, the names of all methods that were pending when the exception hit. The first line of the stack trace is the method that actually generated the exception. The last line of the stack trace is a line in `main`. Often, the exception was thrown by a method that is in the standard library. Look for the first line in your code that appears in the exception report. For example, skip the line that refers to

```
java.lang.String.substring(String.java:1444)
```

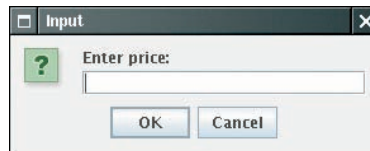
The next line in our example mentions a line number in your code, `Homework1.java`. Once you have the line number in your code, open up the file, go to that line, and look at it! Also look at the name of the exception. In most cases, these two pieces of information will make it completely obvious what went wrong, and you can easily fix your error.

Special Topic 4.4



Using Dialog Boxes for Input and Output

Most program users find the console window rather old-fashioned. The easiest alternative is to create a separate pop-up window for each input.



An Input Dialog Box

Call the static `showInputDialog` method of the `JOptionPane` class, and supply the string that prompts the input from the user. For example,

```
String input = JOptionPane.showInputDialog("Enter price:");
```

That method returns a `String` object. Of course, often you need the input as a number. Use the `Integer.parseInt` and `Double.parseDouble` methods to convert the string to a number:

```
double price = Double.parseDouble(input);
```

FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download a complete program that uses option panes for input and output.

You can also display output in a dialog box:

```
JOptionPane.showMessageDialog(null, "Price: " + price);
```



Computing & Society 4.2 International Alphabets and Unicode

The English alphabet is pretty simple: upper- and lowercase *a* to *z*. Other European languages have accent marks and special characters. For example, German has three so-called *umlaut* characters, *ä*, *ö*, *ü*, and a *double-s* character *ß*. These are not optional frills; you couldn't write a page of German text without using these characters a few times. German keyboards have keys for these characters.



The German Keyboard Layout

Many countries don't use the Roman script at all. Russian, Greek, Hebrew,

Arabic, and Thai letters, to name just a few, have completely different shapes. To complicate matters, Hebrew and Arabic are typed from right to left. Each of these alphabets has about as many characters as the English alphabet.



Hebrew, Arabic, and English

The Chinese languages as well as Japanese and Korean use Chinese characters. Each character represents an idea or thing. Words are made up of one or more of these ideographic characters. Over 70,000 ideographs are known.

Starting in 1987, a consortium of hardware and software manufacturers developed a uniform encoding

scheme called Unicode that is capable of encoding text in essentially all written languages of the world. An early version of Unicode used 16 bits for each character. The Java *char* type corresponds to that encoding.

Today Unicode has grown to a 21-bit code, with definitions for over 100,000 characters (www.unicode.org). There are even plans to add codes for extinct languages, such as Egyptian hieroglyphics. Unfortunately, that means that a Java *char* value does not always correspond to a Unicode character. Some characters in languages such as Chinese or ancient Egyptian occupy two *char* values.



The Chinese Script

CHAPTER SUMMARY

Choose appropriate types for representing numeric data.

- Java has eight primitive types, including four integer types and two floating-point types.
- A numeric computation overflows if the result falls outside the range for the number type.
- Rounding errors occur when an exact conversion between numbers is not possible.
- A *final* variable is a constant. Once its value has been set, it cannot be changed.
- Use named constants to make your programs easier to read and maintain.

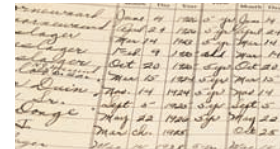


Write arithmetic expressions in Java.

- Mixing integers and floating-point values in an arithmetic expression yields a floating-point value.
- The ++ operator adds 1 to a variable; the -- operator subtracts 1.
- If both arguments of / are integers, the remainder is discarded.
- The % operator computes the remainder of an integer division.
- The Java library declares many mathematical functions, such as Math.sqrt (square root) and Math.pow (raising to a power).
- You use a cast (*typeName*) to convert a value to a different type.

Write programs that read user input and print formatted output.

- Use the Scanner class to read keyboard input in a console window.
- Use the printf method to specify how values should be formatted.

**Carry out hand calculations when developing an algorithm.**

- Pick concrete values for a typical situation to use in a hand calculation.

Write programs that process strings.

- Strings are sequences of characters.
- The length method yields the number of characters in a string.
- Use the + operator to *concatenate* strings; that is, to put them together to yield a longer string.
- Whenever one of the arguments of the + operator is a string, the other argument is converted to a string.
- Use the next method of the Scanner class to read a string containing a single word.
- String positions are counted starting with 0.
- Use the substring method to extract a part of a string.

**STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER**

java.io.PrintStream	exp	java.lang.String	java.math.BigInteger
printf	log	charAt	add
java.lang.Double	log10	length	multiply
parseDouble	max	substring	subtract
java.lang.Integer	min	java.lang.System	java.util.Scanner
MAX_VALUE	pow	in	next
MIN_VALUE	round	java.math.BigDecimal	nextDouble
parseInt	sin	add	nextInt
java.lang.Math	sqr	multiply	javax.swing.JOptionPane
PI	tan	subtract	showInputDialog
abs	toDegrees		showMessageDialog
cos	toRadians		

REVIEW QUESTIONS

- **R4.1** Write declarations for storing the following quantities. Choose between integers and floating-point numbers. Declare constants when appropriate.
 - a. The number of days per week
 - b. The number of days until the end of the semester
 - c. The number of centimeters in an inch
 - d. The height of the tallest person in your class, in centimeters

- **R4.2** What is the value of `mystery` after this sequence of statements?

```
int mystery = 1;
mystery = 1 - 2 * mystery;
mystery = mystery + 1;
```

- **R4.3** What is wrong with the following sequence of statements?

```
int mystery = 1;
mystery = mystery + 1;
int mystery = 1 - 2 * mystery;
```

- **R4.4** Write the following Java expressions in mathematical notation.

- a. `dm = m * (Math.sqrt(1 + v / c) / Math.sqrt(1 - v / c) - 1);`
- b. `volume = Math.PI * r * r * h;`
- c. `volume = 4 * Math.PI * Math.pow(r, 3) / 3;`
- d. `z = Math.sqrt(x * x + y * y);`

- **R4.5** Write the following mathematical expressions in Java.

$$s = s_0 + v_0 t + \frac{1}{2} g t^2$$

$$G = 4\pi^2 \frac{a^3}{p^2(m_1 + m_2)}$$

$$FV = PV \cdot \left(1 + \frac{\text{INT}}{100}\right)^{\text{YRS}}$$

$$c = \sqrt{a^2 + b^2 - 2ab \cos \gamma}$$

- **R4.6** What are the values of the following expressions? In each line, assume that

```
double x = 2.5;
double y = -1.5;
int m = 18;
int n = 4;
```

- a. `x + n * y - (x + n) * y`
- b. `m / n + m % n`
- c. `5 * x - n / 5`
- d. `1 - (1 - (1 - (1 - (1 - n))))`
- e. `Math.sqrt(Math.sqrt(n))`

- **R4.7** What are the values of the following expressions, assuming that *n* is 17 and *m* is 18?

- a. $n / 10 + n \% 10$
- b. $n \% 2 + m \% 2$
- c. $(m + n) / 2$
- d. $(m + n) / 2.0$
- e. `(int) (0.5 * (m + n))`
- f. `(int) Math.round(0.5 * (m + n))`

- ■ **R4.8** What are the values of the following expressions? In each line, assume that

- ```
String s = "Hello";
String t = "World";
```
- a. `s.length() + t.length()`
  - b. `s.substring(1, 2)`
  - c. `s.substring(s.length() / 2, s.length())`
  - d. `s + t`
  - e. `t + s`

- **R4.9** Find at least five *compile-time* errors in the following program.

```
public class HasErrors
{
 public static void main();
 {
 System.out.print(Please enter two numbers:)
 x = in.readDouble;
 y = in.readDouble;
 System.out.println("The sum is " + x + y);
 }
}
```

- ■ **R4.10** Find three *run-time* errors in the following program.

```
public class HasErrors
{
 public static void main(String[] args)
 {
 int x = 0;
 int y = 0;
 Scanner in = new Scanner("System.in");
 System.out.print("Please enter an integer:");
 x = in.readInt();
 System.out.print("Please enter another integer: ");
 x = in.readInt();
 System.out.println("The sum is " + x + y);
 }
}
```

- ■ **R4.11** Consider the following code:

```
CashRegister register = new CashRegister();
register.recordPurchase(19.93);
register.receivePayment(20, 0, 0, 0, 0);
System.out.print("Change: ");
System.out.println(register.giveChange());
```

The code segment prints the total as 0.070000000000000028. Explain why. Give a recommendation to improve the code so that users will not be confused.



- **R4.12** Explain the differences between 2, 2.0, '2', "2", and "2.0".
- **R4.13** Explain what each of the following program segments computes.

**a.** `x = 2;`  
`y = x + x;`

**b.** `s = "2";`  
`t = s + s;`

- **R4.14** Write pseudocode for a program that reads a word and then prints the first character, the last character, and the characters in the middle. For example, if the input is Harry, the program prints H y arr.
- **R4.15** Write pseudocode for a program that reads a name (such as Harold James Morgan) and then prints a monogram consisting of the initial letters of the first, middle, and last name (such as HJM).
- **R4.16** Write pseudocode for a program that computes the first and last digit of a number. For example, if the input is 23456, the program should print 2 and 6. *Hint:* %, Math.log10.



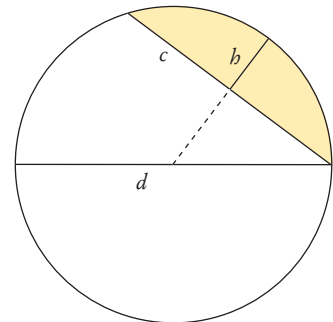
- **R4.17** Modify the pseudocode for the program in How To 4.1 so that the program gives change in quarters, dimes, and nickels. You can assume that the price is a multiple of 5 cents. To develop your pseudocode, first work with a couple of specific values.

- **R4.18** A cocktail shaker is composed of three cone sections. Using realistic values for the radii and heights, compute the total volume, using the formula given in Self Check 21 for a cone section. Then develop an algorithm that works for arbitrary dimensions.

- **R4.19** You are cutting off a piece of pie like this, where  $c$  is the length of the straight part (called the chord length) and  $h$  is the height of the piece. There is an approximate formula for the area:

$$A \approx \frac{2}{3}cb + \frac{b^3}{2c}$$

However,  $h$  is not so easy to measure, whereas the diameter  $d$  of a pie is usually well-known. Calculate the area where the diameter of the pie is 12 inches and the chord length of the segment is 10 inches. Generalize to an algorithm that yields the area for any diameter and chord length.



- **R4.20** The following pseudocode describes how to obtain the name of a day, given the day number (0 = Sunday, 1 = Monday, and so on.)

**Declare a string called names containing "SunMonTueWedThuFriSat".**  
**Compute the starting position as 3 x the day number.**  
**Extract the substring of names at the starting position with length 3.**

Check this pseudocode, using the day number 4. Draw a diagram of the string that is being computed, similar to Figure 3.

- **R4.21** The following pseudocode describes how to swap two letters in a word.

**We are given a string str and two positions i and j. (i comes before j)**  
**Set first to the substring from the start of the string to the last position before i.**

Set *middle* to the substring from positions  $i + 1$  to  $j - 1$ .

Set *last* to the substring from position  $j + 1$  to the end of the string.

Concatenate the following five strings: first, the string containing just the character at position *j*, *middle*, the string containing just the character at position *i*, and *last*.

Check this pseudocode, using the string "Gateway" and positions 2 and 4. Draw a diagram of the string that is being computed, similar to Figure 3.

- ■ **R4.22** How do you get the first character of a string? The last character? How do you remove the first character? The last character?

- ■ ■ **R4.23** Write a program that prints the values

```
3 * 1000 * 1000 * 1000
3.0 * 1000 * 1000 * 1000
```

Explain the results.

## PRACTICE EXERCISES

- **E4.1** Write a program that displays the dimensions of a letter-size ( $8.5 \times 11$  inches) sheet of paper in millimeters. There are 25.4 millimeters per inch. Use constants and comments in your program.
- **E4.2** Write a program that computes and displays the perimeter of a letter-size ( $8.5 \times 11$  inches) sheet of paper and the length of its diagonal.
- **E4.3** Write a program that reads a number and displays the square, cube, and fourth power. Use the `Math.pow` method only for the fourth power.
- ■ **E4.4** Write a program that prompts the user for two integers and then prints
  - The sum
  - The difference
  - The product
  - The average
  - The distance (absolute value of the difference)
  - The maximum (the larger of the two)
  - The minimum (the smaller of the two)

*Hint:* The `max` and `min` functions are declared in the `Math` class.

- ■ **E4.5** Enhance the output of Exercise E4.4 so that the numbers are properly aligned:

```
Sum: 45
Difference: -5
Product: 500
Average: 22.50
Distance: 5
Maximum: 25
Minimum: 20
```

- ■ **E4.6** Write a program that prompts the user for a measurement in meters and then converts it to miles, feet, and inches.

- **E4.7** Write a program that prompts the user for a radius and then prints
  - The area and circumference of a circle with that radius
  - The volume and surface area of a sphere with that radius
- **E4.8** Write a program that asks the user for the lengths of a rectangle's sides. Then print
  - The area and perimeter of the rectangle
  - The length of the diagonal (use the Pythagorean theorem)
- **E4.9** Improve the program discussed in How To 4.1 to allow input of quarters in addition to bills.
- **E4.10** Write a program that asks the user to input
  - The number of gallons of gas in the tank
  - The fuel efficiency in miles per gallon
  - The price of gas per gallon

Then print the cost per 100 miles and how far the car can go with the gas in the tank.

- **E4.11** *File names and extensions.* Write a program that prompts the user for the drive letter (C), the path (\Windows\System), the file name (Readme), and the extension (txt). Then print the complete file name C:\Windows\System\Readme.txt. (If you use UNIX or a Macintosh, skip the drive name and use / instead of \ to separate directories.)
- **E4.12** Write a program that reads a number between 1,000 and 999,999 from the user, where the user enters a comma in the input. Then print the number without a comma. Here is a sample dialog; the user input is in color:

Please enter an integer between 1,000 and 999,999: 23,456  
23456

*Hint:* Read the input as a string. Measure the length of the string. Suppose it contains  $n$  characters. Then extract substrings consisting of the first  $n - 4$  characters and the last three characters.

- **E4.13** Write a program that reads a number between 1,000 and 999,999 from the user and prints it with a comma separating the thousands. Here is a sample dialog; the user input is in color:

Please enter an integer between 1000 and 999999: 23456  
23,456

- **E4.14** *Printing a grid.* Write a program that prints the following grid to play tic-tac-toe.

```
+---+---+
| | |
+---+---+
| | |
+---+---+
| | |
+---+---+
```

Of course, you could simply write seven statements of the form

```
System.out.println("+---+---+");
```

You should do it the smart way, though. Declare string variables to hold two kinds of patterns: a comb-shaped pattern and the bottom line. Print the comb three times and the bottom line once.

- ■ **E4.15** Write a program that reads in an integer and breaks it into a sequence of individual digits. For example, the input 16384 is displayed as

1 6 3 8 4

You may assume that the input has no more than five digits and is not negative.

- ■ **E4.16** Write a program that reads two times in military format (0900, 1730) and prints the number of hours and minutes between the two times. Here is a sample run. User input is in color.

Please enter the first time: 0900  
Please enter the second time: 1730  
8 hours 30 minutes

Extra credit if you can deal with the case where the first time is later than the second:

Please enter the first time: 1730  
Please enter the second time: 0900  
15 hours 30 minutes

- ■ ■ **E4.17** *Writing large letters.* A large letter H can be produced like this:

```
* *
* *

* *
* *
```

It can be declared as a string literal like this:

```
final string LETTER_H = "* *\n* *\n*****\n* *\n* *";
```

(The `\n` escape sequence denotes a “newline” character that causes subsequent characters to be printed on a new line.) Do the same for the letters E, L, and O. Then write the message

```
H
E
L
L
O
```

in large letters.

- ■ **E4.18** Write a program that transforms numbers 1, 2, 3, ..., 12 into the corresponding month names January, February, March, ..., December. *Hint:* Make a very long string “January February March ...”, in which you add spaces such that each month name has *the same length*. Then use substring to extract the month you want.

- ■ **E4.19** Write a program that prints a Christmas tree:

```
 /\
 /\
 /\

 " "
 " "
 " "
```

Remember to use escape sequences.



**E4.20** Enhance the `CashRegister` class by adding separate methods `enterDollars`, `enterQuarters`, `enterDimes`, `enterNickels`, and `enterPennies`.

Use this tester class:

```
public class CashRegisterTester
{
 public static void main (String[] args)
 {
 CashRegister register = new CashRegister();
 register.recordPurchase(20.37);
 register.enterDollars(20);
 register.enterQuarters(2);
 System.out.println("Change: " + register.giveChange());
 System.out.println("Expected: 0.13");
 }
}
```

■ ■ **E4.21** Implement a class `IceCreamCone` with methods `getSurfaceArea()` and `getVolume()`. In the constructor, supply the height and radius of the cone. Be careful when looking up the formula for the surface area—you should only include the outside area along the side of the cone since the cone has an opening on the top to hold the ice cream.

■ ■ **E4.22** Implement a class `SodaCan` whose constructor receives the height and diameter of the soda can. Supply methods `getVolume` and `getSurfaceArea`. Supply a `SodaCanTester` class that tests your class.

■ ■ ■ **E4.23** Implement a class `Balloon` that models a spherical balloon that is being filled with air. The constructor constructs an empty balloon. Supply these methods:

- `void addAir(double amount)` adds the given amount of air
- `double getVolume()` gets the current volume
- `double getSurfaceArea()` gets the current surface area
- `double getRadius()` gets the current radius

Supply a `BalloonTester` class that constructs a balloon, adds  $100 \text{ cm}^3$  of air, tests the three accessor methods, adds another  $100 \text{ cm}^3$  of air, and tests the accessor methods again.

## PROGRAMMING PROJECTS

■ ■ ■ **P4.1** Write a program that helps a person decide whether to buy a hybrid car. Your program's inputs should be:

- The cost of a new car
- The estimated miles driven per year
- The estimated gas price
- The efficiency in miles per gallon
- The estimated resale value after 5 years

Compute the total cost of owning the car for five years. (For simplicity, we will not take the cost of financing into account.)



Obtain realistic prices for a new and used hybrid and a comparable car from the Web. Run your program twice, using today's gas price and 15,000 miles per year. Include pseudocode and the program runs with your assignment.

- ■ P4.2 Easter Sunday is the first Sunday after the first full moon of spring. To compute the date, you can use this algorithm, invented by the mathematician Carl Friedrich Gauss in 1800:

1. Let  $y$  be the year (such as 1800 or 2001).
2. Divide  $y$  by 19 and call the remainder  $a$ . Ignore the quotient.
3. Divide  $y$  by 100 to get a quotient  $b$  and a remainder  $c$ .
4. Divide  $b$  by 4 to get a quotient  $d$  and a remainder  $e$ .
5. Divide  $8 * b + 13$  by 25 to get a quotient  $g$ . Ignore the remainder.
6. Divide  $19 * a + b - d - g + 15$  by 30 to get a remainder  $h$ . Ignore the quotient.
7. Divide  $c$  by 4 to get a quotient  $j$  and a remainder  $k$ .
8. Divide  $a + 11 * h$  by 319 to get a quotient  $m$ . Ignore the remainder.
9. Divide  $2 * e + 2 * j - k - h + m + 32$  by 7 to get a remainder  $r$ . Ignore the quotient.
10. Divide  $h - m + r + 90$  by 25 to get a quotient  $n$ . Ignore the remainder.
11. Divide  $h - m + r + n + 19$  by 32 to get a remainder  $p$ . Ignore the quotient.

Then Easter falls on day  $p$  of month  $n$ . For example, if  $y$  is 2001:

|                 |                |          |
|-----------------|----------------|----------|
| $a = 6$         | $h = 18$       | $n = 4$  |
| $b = 20, c = 1$ | $j = 0, k = 1$ | $p = 15$ |
| $d = 5, e = 0$  | $m = 0$        |          |
| $g = 6$         | $r = 6$        |          |

Therefore, in 2001, Easter Sunday fell on April 15. Write a program that prompts the user for a year and prints out the month and day of Easter Sunday.

- ■ ■ P4.3 In this project, you will perform calculations with triangles. A triangle is defined by the  $x$ - and  $y$ -coordinates of its three corner points.

Your job is to compute the following properties of a given triangle:

- the lengths of all sides
- the angles at all corners
- the perimeter
- the area

Implement a `Triangle` class with appropriate methods. Supply a program that prompts a user for the corner point coordinates and produces a nicely formatted table of the triangle properties.

- ■ ■ P4.4 The `CashRegister` class has an unfortunate limitation: It is closely tied to the coin system in the United States and Canada. Research the system used in most of Europe. Your goal is to produce a cash register that works with euros and cents. Rather than designing another limited `CashRegister` implementation for the European market, you should design a separate `Coin` class and a cash register that can work with coins of all types.

- ■ Business P4.5 The following pseudocode describes how a bookstore computes the price of an order from the total price and the number of the books that were ordered.

Read the total book price and the number of books.  
 Compute the tax (7.5 percent of the total book price).  
 Compute the shipping charge (\$2 per book).  
 The price of the order is the sum of the total book price, the tax, and the shipping charge.  
 Print the price of the order.

Translate this pseudocode into a Java program.

- ■ **Business P4.6** The following pseudocode describes how to turn a string containing a ten-digit phone number (such as "4155551212") into a more readable string with parentheses and dashes, like this: "(415) 555-1212".

Take the substring consisting of the first three characters and surround it with "(" and ")". This is the area code.

Concatenate the area code, the substring consisting of the next three characters, a hyphen, and the substring consisting of the last four characters. This is the formatted number.

Translate this pseudocode into a Java program that reads a telephone number into a string variable, computes the formatted number, and prints it.

- ■ **Business P4.7** The following pseudocode describes how to extract the dollars and cents from a price given as a floating-point value. For example, a price 2.95 yields values 2 and 95 for the dollars and cents.

Assign the price to an integer variable dollars.

Multiply the difference price - dollars by 100 and add 0.5.

Assign the result to an integer variable cents.

Translate this pseudocode into a Java program. Read a price and print the dollars and cents. Test your program with inputs 2.95 and 4.35.

- ■ **Business P4.8** *Giving change.* Implement a program that directs a cashier how to give change. The program has two inputs: the amount due and the amount received from the customer. Display the dollars, quarters, dimes, nickels, and pennies that the customer should receive in return. In order to avoid roundoff errors, the program user should supply both amounts in pennies, for example 274 instead of 2.74.



- ■ **Business P4.9** An online bank wants you to create a program that shows prospective customers how their deposits will grow. Your program should read the initial balance and the annual interest rate. Interest is compounded monthly. Print out the balances after the first three months. Here is a sample run:

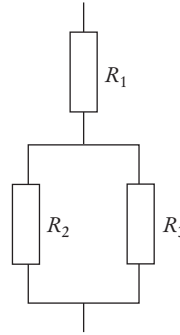
```
Initial balance: 1000
Annual interest rate in percent: 6.0
After first month: 1005.00
After second month: 1010.03
After third month: 1015.08
```

- ■ **Business P4.10** A video club wants to reward its best members with a discount based on the member's number of movie rentals and the number of new members referred by the member. The discount is in percent and is equal to the sum of the rentals and the referrals, but it cannot exceed 75 percent. (*Hint:* Math.min.) Write a program DiscountCalculator to calculate the value of the discount.

Here is a sample run:

```
Enter the number of movie rentals: 56
Enter the number of members referred to the video club: 3
The discount is equal to: 59.00 percent.
```

- **Science P4.11** Consider the following circuit.



Write a program that reads the resistances of the three resistors and computes the total resistance, using Ohm's law.

- ■ **Science P4.12** The dew point temperature  $T_d$  can be calculated (approximately) from the relative humidity  $RH$  and the actual temperature  $T$  by

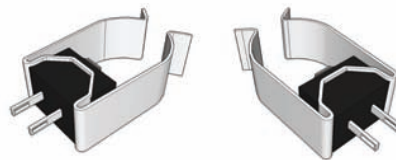
$$T_d = \frac{b \cdot f(T, RH)}{a - f(T, RH)}$$

$$f(T, RH) = \frac{a \cdot T}{b + T} + \ln(RH)$$

where  $a = 17.27$  and  $b = 237.7^\circ \text{C}$ .

Write a program that reads the relative humidity (between 0 and 1) and the temperature (in degrees C) and prints the dew point value. Use the Java function `log` to compute the natural logarithm.

- ■ ■ **Science P4.13** The pipe clip temperature sensors shown here are robust sensors that can be clipped directly onto copper pipes to measure the temperature of the liquids in the pipes.



Each sensor contains a device called a *thermistor*. Thermistors are semiconductor devices that exhibit a temperature-dependent resistance described by:

$$R = R_0 e^{\beta \left( \frac{1}{T} - \frac{1}{T_0} \right)}$$

where  $R$  is the resistance (in  $\Omega$ ) at the temperature  $T$  (in  $^\circ \text{K}$ ), and  $R_0$  is the resistance (in  $\Omega$ ) at the temperature  $T_0$  (in  $^\circ \text{K}$ ).  $\beta$  is a constant that depends on the material used



to make the thermistor. Thermistors are specified by providing values for  $R_0$ ,  $T_0$ , and  $\beta$ .

The thermistors used to make the pipe clip temperature sensors have  $R_0 = 1075 \Omega$  at  $T_0 = 85^\circ\text{C}$ , and  $\beta = 3969^\circ\text{K}$ . (Notice that  $\beta$  has units of  $^\circ\text{K}$ . Recall that the temperature in  $^\circ\text{K}$  is obtained by adding 273 to the temperature in  $^\circ\text{C}$ .) The liquid temperature, in  $^\circ\text{C}$ , is determined from the resistance  $R$ , in  $\Omega$ , using

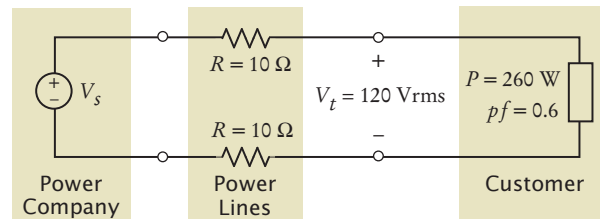
$$T = \frac{\beta T_0}{T_0 \ln\left(\frac{R}{R_0}\right) + \beta} - 273$$

Write a Java program that prompts the user for the thermistor resistance  $R$  and prints a message giving the liquid temperature in  $^\circ\text{C}$ .

#### ■■■ Science P4.24

The circuit shown below illustrates some important aspects of the connection between a power company and one of its customers. The customer is represented by three parameters,  $V_t$ ,  $P$ , and  $pf$ .  $V_t$  is the voltage accessed by plugging into a wall outlet. Customers depend on having a dependable value of  $V_t$  in order for their appliances to work properly. Accordingly, the power company regulates the value of  $V_t$  carefully.

$P$  describes the amount of power used by the customer and is the primary factor in determining the customer's electric bill. The power factor,  $pf$ , is less familiar. (The power factor is calculated as the cosine of an angle so that its value will always be between zero and one.) In this problem you will be asked to write a Java program to investigate the significance of the power factor.

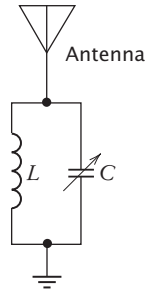


In the figure, the power lines are represented, somewhat simplistically, as resistances in Ohms. The power company is represented as an AC voltage source. The source voltage,  $V_s$ , required to provide the customer with power  $P$  at voltage  $V_t$  can be determined using the formula

$$V_s = \sqrt{\left(V_t + \frac{2RP}{V_t}\right)^2 + \left(\frac{2RP}{pf V_t}\right)^2 (1 - pf^2)}$$

( $V_s$  has units of Vrms.) This formula indicates that the value of  $V_s$  depends on the value of  $pf$ . Write a Java program that prompts the user for a power factor value and then prints a message giving the corresponding value of  $V_s$ , using the values for  $P$ ,  $R$ , and  $V_t$  shown in the figure above.

- ■ ■ **Science P4.25** Consider the following tuning circuit connected to an antenna, where  $C$  is a variable capacitor whose capacitance ranges from  $C_{\min}$  to  $C_{\max}$ .



The tuning circuit selects the frequency  $f = \frac{2\pi}{\sqrt{LC}}$ . To design this circuit for a given frequency, take  $C = \sqrt{C_{\min} C_{\max}}$  and calculate the required inductance  $L$  from  $f$  and  $C$ . Now the circuit can be tuned to any frequency in the range  $f_{\min} = \frac{2\pi}{\sqrt{LC_{\max}}}$  to  $f_{\max} = \frac{2\pi}{\sqrt{LC_{\min}}}$ .

Write a Java program to design a tuning circuit for a given frequency, using a variable capacitor with given values for  $C_{\min}$  and  $C_{\max}$ . (A typical input is  $f = 16.7$  MHz,  $C_{\min} = 14$  pF, and  $C_{\max} = 365$  pF.) The program should read in  $f$  (in Hz),  $C_{\min}$  and  $C_{\max}$  (in F), and print the required inductance value and the range of frequencies to which the circuit can be tuned by varying the capacitance.

- **Science P4.26** According to the Coulomb force law, the electric force between two charged particles of charge  $Q_1$  and  $Q_2$  Coulombs, that are a distance  $r$  meters apart, is  $F = \frac{Q_1 Q_2}{4\pi\epsilon r^2}$  Newtons, where  $\epsilon = 8.854 \times 10^{-12}$  Farads/meter. Write a program that calculates the force on a pair of charged particles, based on the user input of  $Q_1$  Coulombs,  $Q_2$  Coulombs, and  $r$  meters, and then computes and displays the electric force.

## ANSWERS TO SELF-CHECK QUESTIONS

1. `int` and `double`.
2. The world's most populous country, China, has about  $1.2 \times 10^9$  inhabitants. Therefore, individual population counts could be held in an `int`. However, the world population is over  $6 \times 10^9$ . If you compute totals or averages of multiple countries, you can exceed the largest `int` value. Therefore, `double` is a better choice. You could also use `long`, but there is no benefit because the exact population of a country is not known at any point in time.
3. The first initialization is incorrect. The right hand side is a value of type `double`, and it is not legal to initialize an `int` variable with a `double` value. The second initialization is correct—an `int` value can always be converted to a `double`.
4. The first declaration is used inside a method, the second inside a class.
5. Two things: You should use a named constant, not the “magic number” 3.14, and 3.14 is not an accurate representation of  $\pi$ .
6. `double interest = balance * percent / 100;`
7. `double sideLength = Math.sqrt(area);`
8. `4 * PI * Math.pow(radius, 3) / 3`  
or `(4.0 / 3) * PI * Math.pow(radius, 3)`,  
but not `(4 / 3) * PI * Math.pow(radius, 3)`
9. 17 and 29
10. It is the second-to-last digit of `n`. For example, if `n` is 1729, then `n / 10` is 172, and `(n / 10) % 10` is 2.
11. `System.out.print("How old are you? ");`  
`int age = in.nextInt();`
12. There is no prompt that alerts the program user to enter the quantity.
13. The second statement calls `nextInt`, not `nextDouble`. If the user were to enter a price such as 1.95, the program would be terminated with an “input mismatch exception”.
14. There is no colon and space at the end of the prompt. A dialog would look like this:  
Please enter the number of cans6
15. The total volume is 10  
There are four spaces between `is` and 10. One space originates from the format string (the space between `s` and `%`), and three spaces are added before 10 to achieve a field width of 5.
16. Here is a simple solution:  

```
System.out.printf("Bottles: %8d\n", bottles);
System.out.printf("Cans: %8d\n", cans);
```

  
Note the spaces after `Cans:`. Alternatively, you can use format specifiers for the strings. You can even combine all output into a single statement:  

```
System.out.printf("%-9s%8d\n%-9s%8d\n",
"Bottles: ", bottles, "Cans:", cans);
```
17. 

```
int pairs = (totalWidth - tileWidth)
/ (2 * tileWidth);
int tiles = 1 + 2 * pairs;
double gap = (totalWidth -
tiles * tileWidth) / 2.0;
```

  
Be sure that `pairs` is declared as an `int`.
18. Now there are groups of four tiles (gray/white/gray/black) following the initial black tile. Therefore, the algorithm is now  
**number of groups = integer part of (total width - tile width) / (4 x tile width)**  
**number of tiles = 1 + 4 x number of groups**  
The formula for the gap is not changed.
19. The answer depends only on whether the row and column numbers are even or odd, so let's first take the remainder after dividing by 2. Then we can enumerate all expected answers:  

| Row % 2 | Column % 2 | Color |
|---------|------------|-------|
| 0       | 0          | 0     |
| 0       | 1          | 1     |
| 1       | 0          | 1     |
| 1       | 1          | 0     |

  
In the first three entries of the table, the color is simply the sum of the remainders. In the fourth entry, the sum would be 2, but we want a zero. We can achieve that by taking another remainder operation:  
**color = ((row % 2) + (column % 2)) % 2**
20. In nine years, the repair costs increased by \$1,400. Therefore, the increase per year is  $\$1,400 / 9 \approx \$156$ . The repair cost in year 3 would be  $\$100 + 2 \times \$156 = \$412$ . The repair cost in year `n` is  $\$100 + n \times \$156$ . To avoid accumulation of roundoff errors, it is actually

a good idea to use the original expression that yielded \$156, that is,

**Repair cost in year  $n$  =  $100 + n \times 1400 / 9$**

- 21.** The pseudocode follows from the equations:

**bottom volume =  $\pi \times r_1^2 \times h_1$**

**top volume =  $\pi \times r_2^2 \times h_2$**

**middle volume =  $\pi \times (r_1^2 + r_1 \times r_2 + r_2^2) \times h_3 / 3$**

**total volume = bottom volume + top volume + middle volume**

Measuring a typical wine bottle yields

$r_1 = 3.6$ ,  $r_2 = 1.2$ ,  $h_1 = 15$ ,  $h_2 = 7$ ,  $h_3 = 6$

(all in centimeters). Therefore,

bottom volume = 610.73

top volume = 31.67

middle volume = 135.72

total volume = 778.12

The actual volume is 750 ml, which is close enough to our computation to give confidence that it is correct.

- 22.** The length is 12. The space counts as a character.

- 23.** `str.substring(8, 12)` or `str.substring(8)`

- 24.** `str = str + "ming";`

- 25.** Hy

- 26.** `String first = in.next();`  
`String middle = in.next();`  
`String last = in.next();`