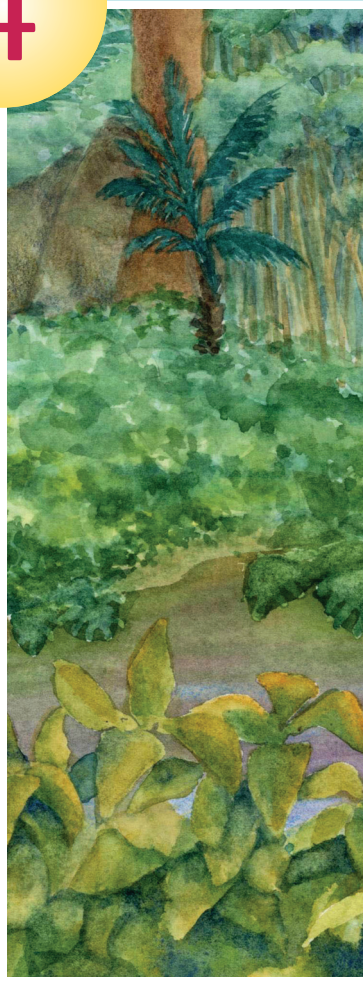
Chapter 4

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 define and manipulate character strings
- To learn how to read program input and produce formatted output

This chapter teaches how to manipulate numbers and character strings in Java. The goal of this chapter is to gain a firm understanding of the fundamental Java data types.

You will learn about the properties and limitations of the number types in Java. You will see how to manipulate numbers and strings in your programs. Finally, we cover the important topic of input and output, which enables you to implement interactive programs.

Chapter Contents

4.1 Number Types

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.

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 integer types has a different range—Advanced Topic 4.2 explains why the range limits are related to powers of two. Generally, you will use the `int` type for integer quantities. However, occasionally, 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
```

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.

Table 1 Primitive Types

Type	Description	Size
`int`	The integer type, with range –2,147,483,648 . . . 2,147,483,647 (about 2 billion)	4 bytes
`byte`	The type describing a single byte, with range –128 . . . 127	1 byte
`short`	The short integer type, with range –32768 . . . 32767	2 bytes
`long`	The long integer type, with range –9,223,372,036,854,775,808 . . . 9,223,372,036,854,775,807	8 bytes
`double`	The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits	8 bytes
`float`	The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits	4 bytes
`char`	The character type, representing code units in the Unicode encoding scheme (see Advanced Topic 4.5)	2 bytes
`boolean`	The type with the two truth values `false` and `true` (see Chapter 6)	1 bit

If you run into this problem, the simplest remedy is to use the `long` type. Advanced Topic 4.1 shows you how to use the arbitary-precision `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 $\pm 10^{308}$ and about 15 significant digits. However, you want to avoid the `float` type—it has less than 7 significant digits. (Some programmers use `float` to save on memory if they need to store a huge set of numbers that do not require much precision.)

Rounding errors occur when an exact conversion between numbers is not possible.

Rounding errors are a more serious issue with floating-point values. Rounding errors can occur when you convert between binary and decimal numbers, or between integers and floating-point numbers. When a value cannot be converted exactly, it is rounded to the nearest match. Consider this example:

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

This problem is caused 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 Advanced Topic 4.2 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 Advanced 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
```

To overcome this problem, you can convert the floating-point value to an integer with a cast:

```
int dollars = (int) balance;
```

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

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.

The cast tells the compiler that you agree to *information loss*, in this case, to the loss of the fractional part. You can also cast to other types, such as `(float)` or `(byte)`.

Use the `Math.round` method to round a floating-point number to the nearest integer.

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.

Syntax 4.1 Cast

(typeName) expression

Example:

```
(int) (balance * 100)
```

Purpose:

To convert an expression to a different type

Self Check

1. Which are the most commonly used number types in Java?
2. When does the cast `(long) x` yield a different result from the call `Math.round(x)`?
3. How do you round the `double` value `x` to the nearest `int` value, assuming that you know that it is less than $2 \cdot 10^9$?

ADVANCED 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 two big integers and how to multiply them.

```
BigInteger a = new BigInteger("1234567890");
BigInteger b = new BigInteger("9876543210");
BigInteger c = a.multiply(b);
System.out.println(c); // Prints 12193263111263526900
```

The `BigDecimal` type carries out floating-point computation 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
```

ADVANCED TOPIC 4.2

Binary Numbers

You are familiar with decimal numbers, which use the digits 0, 1, 2, . . . , 9. Each digit has a place value of 1, 10, $100 = 10^2$, $1000 = 10^3$, and so on. For example,

$$435 = 4 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$$

Fractional digits have place values with negative powers of ten: $0.1 = 10^{-1}$, $0.01 = 10^{-2}$, and so on. For example,

$$4.35 = 4 \cdot 10^0 + 3 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

Computers use binary numbers instead, which have just two digits (0 and 1) and place values that are powers of 2. Binary numbers are easier for computers to manipulate, because it is easier to build logic circuits that differentiate between "off" and "on" than it is to build circuits that can accurately tell ten different voltage levels apart.

It is easy to transform a binary number into a decimal number. Just compute the powers of two that correspond to ones in the binary number. For example,

$$1101 \text{ binary} = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 13$$

Fractional binary numbers use negative powers of two. For example,

$$1.101 \text{ binary} = 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1 + 0.5 + 0.125 = 1.625$$

Converting decimal numbers to binary numbers is a little trickier. Here is an algorithm that converts a decimal integer into its binary equivalent: Keep dividing the integer by 2, keeping track of the remainders. Stop when the number is 0. Then write the remainders as a binary number, starting with the last one. For example,

$$100 \div 2 = 50 \text{ remainder } 0$$
$$50 \div 2 = 25 \text{ remainder } 0$$
$$25 \div 2 = 12 \text{ remainder } 1$$
$$12 \div 2 = 6 \text{ remainder } 0$$
$$6 \div 2 = 3 \text{ remainder } 0$$
$$3 \div 2 = 1 \text{ remainder } 1$$
$$1 \div 2 = 0 \text{ remainder } 1$$

Therefore, 100 in decimal is 1100100 in binary.

To convert a fractional number <1 to its binary format, keep multiplying by 2. If the result is >1, subtract 1. Stop when the number is 0. Then use the digits before the decimal points as the binary digits of the fractional part, starting with the first one. For example,

$$0.35 \cdot 2 = 0.7$$
$$0.7 \cdot 2 = 1.4$$
$$0.4 \cdot 2 = 0.8$$
$$0.8 \cdot 2 = 1.6$$
$$0.6 \cdot 2 = 1.2$$
$$0.2 \cdot 2 = 0.4$$

Here the pattern repeats. That is, the binary representation of 0.35 is 0.01 0110 0110 0110 . . .

To convert any floating-point number into binary, convert the whole part and the fractional part separately. For example, 4.35 is 100.01 0110 0110 0110 . . . in binary.

You don't actually need to know about binary numbers to program in Java, but at times it can be helpful to understand a little about them. For example, knowing that an `int` is represented as a 32-bit binary number explains why the largest integer that you can represent in Java is 0111 1111 1111 1111 1111 1111 1111 1111 binary = 2,147,483,647 decimal. (The first bit is the sign bit. It is off for positive values.)

To convert an integer into its binary representation, you can use the static `toString` method of the `Integer` class. The call `Integer.toString(n, 2)` returns a string with the binary digits of the integer `n`. Conversely, you can convert a string containing binary digits into an integer with the call `Integer.parseInt(digitString, 2)`. In both of these method calls, the second parameter denotes the base of the number system. It can be any number between 0 and 36. You can use these two methods to convert between decimal and binary integers. However, the Java library has no convenient method to do the same for floating-point numbers.

Now you can see why we had to fight with a roundoff error when computing 100 times 4.35. If you actually carry out the long multiplication, you get:

```
1 1 0 0 1 0 0 * 1 0 0.0 1|0 1 1 0|0 1 1 0|0 1 1 0 ...

1 0 0.0 1|0 1 1 0|0 1 1 0|0 1 1 0 ...
  1 0 0.0 1|0 1 1 0|0 1 1 0|0 1 1 ...
    0
      0
        1 0 0.0 1|0 1 1 0|0 1 1 0 ...
          0
            0
_____________________________________
1 1 0 1 1 0 0 1 0.1 1 1 1 1 1 1 1 ...
```

That is, the result is 434, followed by an infinite number of 1s. The fractional part of the product is the binary equivalent of an infinite decimal fraction 0.999999 . . . , which is equal to 1. But the CPU can store only a finite number of 1s, and it discards some of them when converting the result to a decimal number.

RANDOM FACT 4.1

The Pentium Floating-Point Bug

In 1994, Intel Corporation released what was then its most powerful processor, the first of the Pentium series. Unlike previous generations of Intel's processors, the Pentium 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 an immediate success.

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 run on the slower 486 processor, which preceded the Pentium in Intel's lineup. This should not have happened. The optimal roundoff behavior of floating-point calculations had been standardized by the Institute of Electrical and Electronics 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 a Pentium processor, however, 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. (Subsequent versions of the Pentium, such as the Pentium III and IV, are free of the problem.) 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 replacing all the Pentium processors that it had already sold would cost it 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 had to cave in to public demand and replaced all defective chips, at a cost of about 475 million dollars.

What do you think? Intel claims that the probability of the bug occurring in any calculation is extremely small—smaller than many chances you take every day, such as driving to work in an automobile. Indeed, many users had used their Pentium computers for many months without reporting any ill effects, and the computations that Professor Nicely was doing are hardly examples of typical user needs. As a result of its public relations blunder, Intel ended up paying a large amount of money. Undoubtedly, some of that money was added to chip prices and thus actually paid by Intel's customers. Also, a large number of processors, whose manufacture consumed energy and caused some environmental impact, were destroyed without benefiting anyone. Could Intel have been justified in wanting to replace only the processors of those users who could reasonably be expected to suffer an impact from the problem?

Suppose that, instead of stonewalling, Intel had offered you the choice of a free replacement processor or a $200 rebate. What would you have done? Would you have replaced your faulty chip, or would you have taken your chances and pocketed the money?

4.2 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;
```

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

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.

In Java, constants are identified with the keyword `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.

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

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;
```

Frequently, constant values are needed in several methods. Then you should declare them together with the instance fields of a class and tag them as `static` and `final`. As before, `final` indicates that the value is a constant. The `static` keyword means that the constant belongs to the class—this is explained in greater detail in Chapter 9.)

```
public class CashRegister
{
   // Methods
   . . .

   // 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 fields
   private double purchase;
   private double payment;
}
```

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 defined, then a period, then the name of the constant, such as `CashRegister.NICKEL_VALUE`.

The `Math` class from the standard library defines 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 of your methods. For example,

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

The sample program at the end of this section 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.enterPayment(1, 2, 1, 1, 4);
```

enters a payment consisting of one dollar, two quarters, one dime, one nickel, and four pennies. The `enterPayment` 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.

SYNTAX 4.2 Constant Definition

In a method:
`final` *typeName variableName* `=` *expression*`;`

In a class:
accessSpecifier `static final` *typeName variableName* `=` *expression*`;`

Example:

```
final double NICKEL_VALUE = 0.05;
public static final double LITERS_PER_GALLON = 3.785;
```

Purpose:

To define a constant in a method or a class

File CashRegister.java

```
/**
   A cash register totals up sales and computes change due.
*/
public class CashRegister
{
   /**
      Constructs a cash register with no money in it.
   */
   public CashRegister()
   {
      purchase = 0;
      payment = 0;
   }

   /**
      Records the purchase price of an item.
      @param amount the price of the purchased item
   */
   public void recordPurchase(double amount)
   {
      purchase = purchase + amount;
   }

   /**
      Enters the payment received from the customer.
      @param dollars the number of dollars in the payment
      @param quarters the number of quarters in the payment
      @param dimes the number of dimes in the payment
      @param nickels the number of nickels in the payment
      @param pennies the number of pennies in the payment
   */
   public void enterPayment(int dollars, int quarters,
         int dimes, int nickels, int pennies)
   {
      payment = dollars + quarters * QUARTER_VALUE + dimes * DIME_VALUE
            + nickels * NICKEL_VALUE + pennies * PENNY_VALUE;
   }

   /**
      Computes the change due and resets the machine for the next customer.
      @return the change due to the customer
   */
   public double giveChange()
   {
      double change = payment - purchase;
      purchase = 0;
      payment = 0;
      return change;
   }

   public static final double QUARTER_VALUE = 0.25;
   public static final double DIME_VALUE = 0.1;
```

```
53     public static final double NICKEL_VALUE = 0.05;
54     public static final double PENNY_VALUE = 0.01;
55
56     private double purchase;
57     private double payment;
58  }
```

File 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.enterPayment(2, 0, 5, 0, 0);
13       System.out.print("Change=");
14       System.out.println(register.giveChange());
15
16       register.recordPurchase(2.25);
17       register.recordPurchase(19.25);
18       register.enterPayment(23, 2, 0, 0, 0);
19       System.out.print("Change=");
20       System.out.println(register.giveChange());
21    }
22 }
```

Output

```
Change=0.25
Change=2.0
```

Self Check

4. 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;
   ```

5. What is wrong with the following statement?

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

Quality 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;
```

By the way, the device

```
final int THREE_HUNDRED_AND_SIXTY_FIVE = 365;
```

is counterproductive and frowned upon.

Quality Tip 4.2

Choose Descriptive Variable Names

In algebra, variable names are usually just one letter long, such as p or A, maybe with a subscript such as p_1. You might be tempted to save yourself a lot of typing by using short variable names in your Java programs:

```
payment = d + q * QV + di * DIV + n * NV + p * PV;
```

Compare this with the following statement:

```
payment = dollars + quarters * QUARTER_VALUE + dimes * DIME_VALUE
      + nickels * NICKEL_VALUE + pennies * PENNY_VALUE;
```

The advantage is obvious. Reading `dollars` is a lot less trouble than reading `d` and then figuring out that it must mean "dollars".

In practical programming, descriptive variable names are particularly important when programs are written by more than one person. It may be obvious to you that `d` stands for dollars, but is it obvious to the person who needs to update your code years later, long after you were promoted (or laid off)? For that matter, will you remember yourself what `d` means when you look at the code six months from now?

4.3 Assignment, Increment, and Decrement

The = operator is called the assignment operator. On the left, you need a variable name. The right-hand side can be a single value or an expression. The assignment operator sets the variable to the given value. So far, that's straightforward. But now let's look at a more interesting use of the assignment operator. Consider the statement

```
items = items + 1;
```

It means, "Compute the value of the expression `items + 1`, and place the result again into the variable `items`." (See Figure 1.) The net effect of executing this statement is to increment `items` by 1. For example, if `items` was 3 before execution of the statement, it is set to 4 afterwards. (This statement would be useful if the cash register kept track of the number of purchased items.)

The = sign does *not* mean that the left-hand side is equal to the right-hand side. Instead, it is an instruction to copy the right-hand-side value into the left-hand-side variable. You should not confuse this assignment operation with the = relation used in algebra to denote equality. The assignment operator is an instruction to do something, namely place a value into a variable. The mathematical equality states the fact that two values are equal. Of course, in mathematics it would make no sense to write that $i = i + 1$; no integer can equal itself plus 1.

Assignment to a variable is not the same as mathematical equality.

The concepts of assignment and equality have no relationship with each other, and it is a bit unfortunate that the Java language (following C and C++) uses = to denote assignment. Other programming languages use a symbol such as <- or :=, which avoids the confusion.

The ++ and -- operators increment and decrement a variable.

The increment operation is so common when writing programs that there is a special shorthand for it, namely

```
items++;
```

This statement also adds 1 to `items`. However, it is easier to type and read than the explicit assignment statement. As you might have guessed, there is also a decrement operator --. The statement

```
items--;
```

subtracts 1 from `items`.

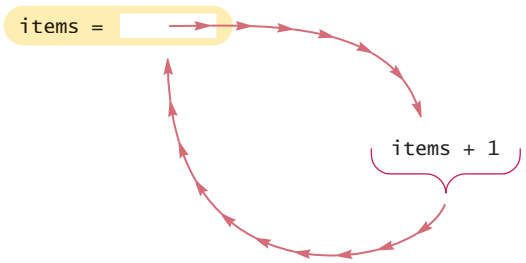


Figure 1
Incrementing a Variable

Self Check

6. What is the meaning of the following statement?

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

7. What is the value of `n` after the following sequence of statements?

```
n--;
n++;
n--;
```

Productivity Hint 4.1

Avoid Unstable Layout

Arrange program code and comments so that the program is easy to read. For example, do not cram all statements on a single line, and make sure that braces `{ }` line up.

However, be careful when you embark on beautification efforts. Some programmers like to line up the = signs in a series of assignments, like this:

```
nickels  = 0;
dimes    = 0;
quarters = 0;
```

This looks very neat, but the layout is not stable. Suppose you add a line like the one at the bottom of this:

```
nickels  = 0;
dimes    = 0;
quarters = 0;
halfDollars = 0;
```

Oops, now the = signs no longer line up, and you have the extra work of lining them up again.

Here is another example. Some programmers like to put a column of asterisks (*) in documentation comments, like this:

```
/**
 * Computes the change due and resets the cash register for the
 * next customer.
 * @return the change due to the customer
 */
```

It looks pretty, but it is tedious to rearrange the asterisks when editing comments.

You may not care about these issues. Perhaps you plan to beautify your program just before it is finished, when you are about to turn in your homework. That is not a particularly useful approach. In practice, programs are never finished. They are continuously improved and updated. It is better to develop the habit of laying out your programs well from the start and keeping them legible at all times. Therefore, it is a good idea to avoid layout schemes that are hard to maintain.

ADVANCED TOPIC 4.3

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,

```
items *= 2;
```

is another way of writing

```
items = items * 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.

4.4 Arithmetic Operations and Mathematical Functions

You already saw how to add, subtract, and multiply values. Division is indicated with a `/`, not a fraction bar. For example,

$$\frac{a + b}{2}$$

becomes

```
(a + b) / 2
```

Parentheses are used just as in algebra: to indicate in which order the subexpressions 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. Just as in regular algebraic notation, multiplication and division bind more strongly than addition and subtraction. For example, in the expression `a + b / 2`, the `/` is carried out first, even though the `+` operation occurs farther to the left.

If both arguments of the `/` operator are integers, the result is an integer and 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 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.

The % operator computes the remainder of a division.

If you are interested only in the remainder of an integer division, 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.

Here is a typical use for the integer / and % operations. Suppose you want to know how much change a cash register should give, using separate values for dollars and cents. You can compute the value as an integer, denominated in cents, and then compute the whole dollar amount and the remaining change:

```
final int PENNIES_PER_NICKEL = 5;
final int PENNIES_PER_DIME = 10;
final int PENNIES_PER_QUARTER = 25;
final int PENNIES_PER_DOLLAR = 100;

// Compute total value in pennies
int total = dollars * PENNIES_PER_DOLLAR + quarters * PENNIES_PER_QUARTER
      + nickels * PENNIES_PER_NICKEL + dimes * PENNIES_PER_DIME + pennies;

// Use integer division to convert to dollars, cents
int dollars = total / PENNIES_PER_DOLLAR;
int cents = total % PENNIES_PER_DOLLAR;
```

For example, if `total` is 243, then `dollars` is set to 2 and `cents` to 43.

The Math class contains methods sqrt and pow to compute square roots and powers.

To compute x^n, you write `Math.pow(x, n)`. However, to compute x^2 it is significantly more efficient simply to compute `x * x`.

To take the square root of a number, you use the `Math.sqrt` method. For example, $\sqrt{x}$ is written as `Math.sqrt(x)`.

In algebra, you use fractions, superscripts for exponents, and radical signs for 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 subexpression

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

of the quadratic formula becomes

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

Figure 2 shows how to analyze such an expression. With complicated expressions like these, it is not always easy to keep the parentheses `( )` matched—see Common Error 4.2.

Table 2 shows additional methods of the `Math` class. Inputs and outputs are floating-point numbers.

Table 2 Mathematical Methods

Function	Returns
`Math.sqrt(x)`	Square root of x (≥ 0)
`Math.pow(x, y)`	x^y ($x > 0$, or $x = 0$ and $y > 0$, or $x < 0$ and y is an integer)
`Math.sin(x)`	Sine of x (x in radians)
`Math.cos(x)`	Cosine of x
`Math.tan(x)`	Tangent of x
`Math.asin(x)`	Arc sine ($\sin^{-1} x \in [-\pi/2, \pi/2]$, $x \in [-1, 1]$)
`Math.acos(x)`	Arc cosine ($\cos^{-1} x \in [0, \pi]$, $x \in [-1, 1]$)
`Math.atan(x)`	Arc tangent ($\tan^{-1} x \in [-\pi/2, \pi/2]$)
`Math.atan2(y, x)`	Arc tangent ($\tan^{-1} y/x \in [-\pi, \pi]$), x may be 0
`Math.toRadians(x)`	Convert x degrees to radians (i.e., returns $x \cdot \pi/180$)
`Math.toDegrees(x)`	Convert x radians to degrees (i.e., returns $x \cdot 180/\pi$)
`Math.exp(x)`	e^x
`Math.log(x)`	Natural log ($\ln(x)$, $x > 0$)
`Math.round(x)`	Closest integer to x (as a `long`)
`Math.ceil(x)`	Smallest integer $\geq x$ (as a `double`)
`Math.floor(x)`	Largest integer $\leq x$ (as a `double`)
`Math.abs(x)`	Absolute value $\lvert x \rvert$
`Math.max(x, y)`	The larger of x and y
`Math.min(x, y)`	The smaller of x and y

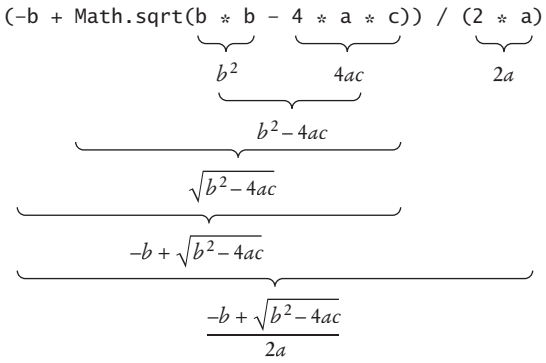


Figure 2 Analyzing an Expression

Self Check

8. What is the value of `1729 / 100`? Of `1729 % 100`?
9. Why doesn't the following statement compute the average of `s1`, `s2`, and `s3`?

   ```
   double average = s1 + s2 + s3 / 3; // Error
   ```

10. What is the value of `Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))` in mathematical notation?

Common Error 4.1

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 program segment that computes the average of three integers.

```
int s1 = 5; // Score of test 1
int s2 = 6; // Score of test 2
int s3 = 3; // Score of test 3
double average = (s1 + s2 + s3) / 3;  // Error
System.out.print("Your average score is ");
System.out.println(average);
```

What could be wrong with that? Of course, the average of `s1`, `s2`, and `s3` is

$$\frac{s_1 + s_2 + s_3}{3}$$

Here, however, the `/` does not mean division in the mathematical sense. It denotes integer division, because the values `s1 + s2 + s3` and `3` are both integers. For example, if the scores add up to 14, the average is computed to be 4, the result of the integer division of 14 by 3.

That integer 4 is then moved into the floating-point variable average. The remedy is to make either the numerator or denominator into a floating-point number:

```
double total = s1 + s2 + s3;
double average = total / 3;
```

or

```
double average = (s1 + s2 + s3) / 3.0;
```

COMMON ERROR 4.2

Unbalanced Parentheses

Consider the expression

```
1.5 * ((-(b - Math.sqrt(b * b - 4 * a * c)) / (2 * a))
```

What is wrong with it? Count the parentheses. There are five opening parentheses (and four closing parentheses). The parentheses are unbalanced. This kind of typing error is very common with complicated expressions. Now consider this expression.

```
1.5 * (Math.sqrt(b * b - 4 * a * c))) - ((b / (2 * a))
```

This expression has five opening parentheses (and five closing parentheses), but it is still not correct. In the middle of the expression,

```
1.5 * (Math.sqrt(b * b - 4 * a * c))) - ((b / (2 * a))
```

there are only two opening parentheses (but three closing parentheses), which is an error. In the middle of an expression, the count of opening parentheses must be greater than or equal to the count of closing parentheses, 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, so keep only one count when scanning the expression. Start with 1 at the first opening parenthesis; add 1 whenever you see an opening parenthesis; subtract 1 whenever you see a closing parenthesis. Say the numbers aloud as you scan the expression. If the count ever drops below zero, or if it is not zero at the end, the parentheses are unbalanced. For example, when scanning the previous expression, you would mutter

```
1.5 * (Math.sqrt(b * b - 4 * a * c) )  ) - ((b / (2 * a))
      1         2                 1 0  –1
```

and you would find the error.

QUALITY TIP 4.3

White Space

The compiler does not care whether you write your entire program onto a single line or place every symbol onto a separate line. The human reader, though, cares very much. You should use blank lines to group your code visually into sections. For example, you can signal to the reader that an output prompt and the corresponding input statement belong together

by inserting a blank line before and after the group. You will find many examples in the source code listings in this book.

White space inside expressions is also important. 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, as in `-b`. That way, it can be easily distinguished from a binary minus, as in `a - b`. Don't put spaces between a method name and the parentheses, but do put a space after every Java keyword. That makes it easy to see that the `sqrt` in `Math.sqrt(x)` is a method name, whereas the `if` in `if (x > 0) . . .` is a keyword.

QUALITY TIP 4.4

Factor Out Common Code

Suppose you want to find both solutions of the quadratic equation $ax^2 + bx + c = 0$. The quadratic formula tells us that the solutions are

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In Java, there is no analog to the ± operation, which indicates how to obtain two solutions simultaneously. Both solutions must be computed separately:

```
x1 = (-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a);
x2 = (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);
```

This approach has two problems. First, the computation of `Math.sqrt(b * b - 4 * a * c)` is carried out twice, which wastes time. Second, whenever the same code is replicated, the possibility of a typing error increases. The remedy is to factor out the common code:

```
double root = Math.sqrt(b * b - 4 * a * c);
x1 = (-b + root) / (2 * a);
x2 = (-b - root) / (2 * a);
```

You could go even further and factor out the computation of `2 * a`, but the gain from factoring out very simple computations is too small to warrant the effort.

4.5 Calling 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 have a special form: they are static methods that do not operate on an object.

That is, you don't call

```
double x = 4;
double root = x.sqrt(); // Error
```

because, in Java, numbers are not objects, so you can never invoke a method on a number. Instead, you pass a number as an explicit parameter to a method, enclosing the number in parentheses after the method name. For example, the number value `x` can be a parameter of the `Math.sqrt` method: `Math.sqrt(x)`.

A static method does not operate on an object.

This call makes it appear as if the `sqrt` method is applied to an object called `Math`, because `Math` precedes `sqrt` just as `harrysChecking` precedes `getBalance` in a method call `harrysChecking.getBalance()`. However, `Math` is a class, not an object. A method such as `Math.round` that does not operate on any object is called a *static* method. (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 defined inside classes. You must specify the class to which the `sqrt` method belongs—hence the call is `Math.sqrt(x)`.

How can you tell whether `Math` is a class or an object? All classes in the Java library start with an uppercase letter (such as `System`). Objects and methods start with a lowercase letter (such as `out` and `println`). (You can tell objects and methods apart because method calls are followed by parentheses.) Therefore, `System.out.println()` denotes a call of the `println` method on the `out` object inside the `System` class. On the other hand, `Math.sqrt(x)` denotes a call to the `sqrt` method inside the `Math` class. This use of upper- and lowercase letters is merely a convention, not a rule of the Java language. It is, however, a convention that the authors of the Java class libraries follow consistently. You should do the same in your programs. If you give names to objects or methods that start with uppercase letters, you will likely confuse your fellow programmers. Therefore, we strongly recommend that you follow the standard naming convention.

SYNTAX 4.3 Static Method Call

ClassName.*methodName*(*parameters*)

Example:

```
Math.sqrt(4)
```

Purpose:

To invoke a static method (a method that does not operate on an object) and supply its parameters

SELF CHECK

11. Why can't you call `x.pow(y)` to compute x^y?

12. Is the call `System.out.println(4)` a static method call?

Common Error 4.3

Roundoff Errors

Roundoff errors are a fact of life when calculating with floating-point numbers. You probably have encountered this phenomenon yourself with manual calculations. If you calculate 1/3 to two decimal places, you get 0.33. Multiplying again by 3, you obtain 0.99, not 1.00.

In the processor hardware, numbers are represented in the binary number system, not in decimal. You still get roundoff errors when binary digits are lost. They just may crop up at different places than you might expect. Here is an example:

```
double f = 4.35;
int n = (int) (100 * f);
System.out.println(n); // Prints 434!
```

Of course, one hundred times 4.35 is 435, but the program prints 434.

Computers represent numbers in the binary system (see Advanced Topic 4.2). In the binary system, there is no exact representation for 4.35, just as there is no exact representation for 1/3 in the decimal system. The representation used by the computer is just a little less than 4.35, so 100 times that value is just a little less than 435. When a floating-point value is converted to an integer, the entire fractional part is discarded, even if it is almost 1. As a result, the integer 434 is stored in `n`. Remedy: Use `Math.round` to convert floating-point numbers to integers. The `round` method returns the *closest* integer.

```
int n = (int) Math.round(100 * f);  // OK, n is 435
```

How To 4.1

Carrying Out Computations

Many programming problems require that you use mathematical formulas to compute values. It is not always obvious how to turn a problem statement into a sequence of mathematical formulas and, ultimately, statements in the Java programming language.

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

For example, suppose you are asked to simulate a postage stamp vending machine. A customer inserts money into the vending machine. Then the customer pushes a "First class stamps" button. The vending machine gives out as many first-class stamps as the customer paid for. (A first-class stamp cost 37 cents at the time this book was written.) Finally, the customer pushes a "Penny stamps" button. The machine gives the change in penny (1-cent) stamps.

In this problem, there is one input:

- The amount of money the customer inserts

There are two desired outputs:

- The number of first-class stamps the machine returns
- The number of penny stamps 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 first-class stamp costs 37 cents and the customer inserts $1.00. That's enough for two stamps (74 cents) but not enough for three stamps ($1.11). Therefore, the machine returns two first-class stamps and 26 penny stamps.

Step 3 Find mathematical equations that compute the answers.

Given an amount of money and the price of a first-class stamp, how can you compute how many first-class stamps can be purchased with the money? Clearly, the answer is related to the quotient

$$\frac{\text{amount of money}}{\text{price of first-class stamp}}$$

For example, suppose the customer paid $1.00. Use a pocket calculator to compute the quotient: $1.00/$0.37 ≈ 2.7027.

How do you get "2 stamps" out of 2.7027? It's the integer part. By discarding the fractional part, you get the number of whole stamps the customer has purchased.

In mathematical notation,

$$\text{number of first-class stamps} = \left\lfloor \frac{\text{money}}{\text{price of first-class stamp}} \right\rfloor$$

where $\lfloor x \rfloor$ denotes the largest integer $\leq x$. That function is sometimes called the "floor function".

You now know how to compute the number of stamps that are given out when the customer pushes the "First-class stamps" button. When the customer gets the stamps, the amount of money is reduced by the value of the stamps purchased. For example, if the customer gets two stamps, the remaining money is $0.26—the difference between $1.00 and 2 · $0.37. Here is the general formula:

$$\text{remaining money} = \text{money} - \text{number of first-class stamps} \cdot \text{price of first-class stamp}$$

How many penny stamps does the remaining money buy? That's easy. If $0.26 is left, the customer gets 26 stamps. In general, the number of penny stamps is

$$\text{number of penny stamps} = 100 \cdot \text{remaining money}$$

Step 4 Turn the mathematical equations into Java statements.

In Java, you can compute the integer part of a nonnegative floating-point value by applying an `(int)` cast. Therefore, you can compute the number of first-class stamps with the following statement:

```
firstClassStamps = (int) (money / FIRST_CLASS_STAMP_PRICE);
money = money - firstClassStamps * FIRST_CLASS_STAMP_PRICE;
```

Finally, the number of penny stamps is

```
pennyStamps = 100 * money;
```

That's not quite right, though. The value of `pennyStamps` should be an integer, but the right-hand side is a floating-point number. Therefore, the correct statement is

```
pennyStamps = (int) Math.round(100 * money);
```

Step 5 Build a class that carries out your computations.

How To 3.1 explains how to develop a class by finding methods and instance variables. In our case, we can find three methods:

- `void insert(double amount)`
- `int giveFirstClassStamps()`
- `int givePennyStamps()`

The state of a vending machine can be described by the amount of money that the customer has available for purchases. Therefore, we supply one instance variable, `money`.

Here is the implementation:

```
public class StampMachine
{
   public StampMachine()
   {
      money = 0;
   }

   public void insert(double amount)
   {
      money = money + amount;
   }

   public int giveFirstClassStamps()
   {
      int firstClassStamps = (int) (money / FIRST_CLASS_STAMP_PRICE);
      money = money - firstClassStamps * FIRST_CLASS_STAMP_PRICE;
      return firstClassStamps;
   }

   public int givePennyStamps()
   {
      int pennyStamps = (int) Math.round(100 * money);
      money = 0;
      return pennyStamps;
   }

   public static final double FIRST_CLASS_STAMP_PRICE = 0.37;
   private double money;
}
```

Step 6 Test your class.

Run a test program (or use an integrated environment such as BlueJ) to verify that the values that your class computes are the same values that you computed by hand. In our example, try the statements

```
StampMachine machine = new StampMachine();
machine.insert(1);
System.out.print("First class stamps: ");
System.out.println(machine.giveFirstClassStamps());
System.out.print("Penny stamps: ");
System.out.println(machine.givePennyStamps());
```

Check that the result is

```
First class stamps: 2
Penny stamps: 26
```

4.6 Strings

Next to numbers, strings are the most important data type that most programs use. A string is a sequence of characters, such as `"Hello, World!"`. In Java, strings are enclosed in quotation marks, which are not themselves part of the string. Note that, unlike numbers, strings are objects. (You can tell that `String` is a class name because it starts with an uppercase letter. The primitive types `int` and `double` start with lowercase letters.)

A string is a sequence of characters. Strings are objects of the `String` class.

The number of characters in a string is called the length of the string. For example, the length of `"Hello, World!"` is 13. You can compute the length of a string with the `length` method.

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

A string of length zero, containing no characters, is called the empty string and is written as `""`.

Use the + operator to put strings together to form a longer string.

```
String name = "Dave";
String message = "Hello, " + name;
```

Strings can be concatenated, that is, put end to end to yield a new longer string. String concatenation is denoted by the + operator.

The + operator concatenates two strings, provided one of the expressions, either 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 a = "Agent";
int n = 7;
String bond = a + n;
```

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

Because `a` is a string, `n` 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 to reduce 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`.

Sometimes you have a string that contains a number, usually from user input. For example, suppose that the string variable `input` has the value `"19"`. To get the integer value 19, you use the static `parseInt` method of the `Integer` class.

```
int count = Integer.parseInt(input);
   // count is the integer 19
```

Figure 3 String Positions

If a string contains the digits of a number, you use the `Integer.parseInt` or `Double.parseDouble` method to obtain the number value.

To convert a string containing floating-point digits to its floating-point value, use the static `parseDouble` method of the `Double` class. For example, suppose `input` is the string `"3.95"`.

```
double price = Double.parseDouble(input);
   // price is the floating-point number 3.95
```

However, if the string contains spaces or other characters that cannot occur inside numbers, an error occurs. For now, we will always assume that user input does not contain invalid characters.

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

The `substring` method computes substrings of a string. The call

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

returns a string that is made up of the characters in the string `s`, 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"
```

String positions are counted starting with 0.

The `substring` operation makes a string that consists of five characters taken from the string `greeting`. A curious aspect of the `substring` operation is the numbering of the starting and ending positions. The first string position is labeled `0`, the second one 1, and so on. For example, Figure 3 shows the position numbers in the `greeting` string.

The position number of the last character (12 for the string `"Hello, World!"`) is always 1 less than the length of the string.

Let us figure out how to extract the substring `"World"`. Count characters starting at `0`, not 1. You find that `W`, the eighth character, has position number 7. The first character that you don't want, `!`, is the character at position 12 (see Figure 4). Therefore, the appropriate substring command is

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

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

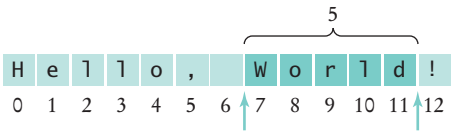


Figure 4 Extracting a Substring

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 second parameter of 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!"`.

If you supply an illegal string position (a negative number, or a value that is larger than the length of the string), then your program terminates with an error message.

In this section, we have made the assumption that each character in a string occupies a single position. Unfortunately, that assumption is not quite correct. If you process strings that contain characters from international alphabets or special symbols, some characters may occupy two positions—see Advanced Topic 4.5.

Self Check

13. Assuming the `String` variable `s` holds the value `"Agent"`, what is the effect of the assignment `s = s + s.length()`?

14. Assuming the `String` variable `river` holds the value `"Mississippi"`, what is the value of `river.substring(1, 2)`? Of `river.substring(2, river.length() - 3)`?

Productivity Hint 4.2

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)
```

An amazing number of students simply give up at that point, saying "it didn't work", or "my program died", without ever reading 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! In the great majority of cases, knowing the name of the exception and the line that caused it make it completely obvious what went wrong, and you can easily fix your error.

ADVANCED TOPIC 4.4

Escape Sequences

Suppose you want to display a string containing quotation marks, such as

```
Hello, "World"!
```

You can't use

```
System.out.println("Hello, "World"!");
```

As soon as the compiler reads `"Hello, "`, it thinks the string is finished, and then it gets all confused about `World` followed by two quotation marks. A human would probably realize that the second and third quotation marks were supposed to be part of the string, but a compiler has a one-track mind. If a simple analysis of the input doesn't make sense to it, it just refuses to go on, and reports an error. Well, how do you then display quotation marks on the screen? You precede the quotation marks inside the string with a *backslash* character. Inside a string, the sequence `\"` denotes a literal quote, not the end of a string. The correct display statement is, therefore

```
System.out.println("Hello, \"World\"!");
```

The backslash character is used as an *escape* character; the character sequence `\"` is called an escape sequence. The backslash does not denote itself; instead, it is used to encode other characters that would otherwise be difficult to include in a string.

Now, what do you do if you actually want to print a backslash (for example, to specify a Windows file name)? You must enter two `\\` in a row, like this:

```
System.out.println("The secret message is in C:\\Temp\\Secret.txt");
```

This statement prints

```
The secret message is in C:\Temp\Secret.txt
```

Another escape sequence occasionally used is `\n`, which denotes a *newline* or line feed 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. Of course, you could have achieved the same effect with three separate calls to `println`.

Finally, escape sequences are useful for including international characters in a string. For example, suppose you want to print "All the way to San José!", with an accented letter (é). If you use a U.S. keyboard, you may not have a key to generate that letter. Java uses the *Unicode* encoding scheme to denote international characters. For example, the é character has Unicode encoding `00E9`. You can include that character inside a string by writing `\u`, followed by its Unicode encoding:

```
System.out.println("All the way to San Jos\u00E9!");
```

You can look up the codes for the U.S. English and Western European characters in Appendix B, and codes for thousands of characters in reference [1].

ADVANCED TOPIC 4.5

Strings and the char Type

Strings are sequences of Unicode characters (see Random Fact 4.2). Character constants look like string constants, except that character constants are delimited by single quotes: `'H'` is a character, `"H"` is a string containing a single character. You can use escape sequences (see Advanced Topic 4.4) inside character constants. For example, `'\n'` is the newline character, and `'\u00E9'` is the character é. You can find the values of the character constants that are used in Western European languages in Appendix B.

Characters have numeric values. For example, if you look at Appendix B, you can see that the character `'H'` is actually encoded as the number 72.

When Java was first designed, each Unicode character was encoded as a two-byte quantity. The `char` type was intended to hold the code of a Unicode character. However, as of 2003, Unicode had grown so large that some characters needed to be encoded as pairs of `char` values. Thus, you can no longer think of a `char` value as a character. Technically speaking, a `char` value is a *code unit* in the UTF-16 encoding of Unicode. That encoding represents the most common characters as a single `char` value, and less common or *supplementary* characters as a pair of `char` values.

The `charAt` method of the `String` class returns a code unit from a string. As with the `substring` method, the positions in the string are counted starting at `0`. For example, the statement

```
String greeting = "Hello";
char ch = greeting.charAt(0);
```

sets `ch` to the value `'H'`.

However, if you use `char` variables, your programs may fail with some strings that contain international or symbolic characters. For example, the single character $\mathbb{Z}$ (the mathematical symbol for the set of integers) is encoded by the two code units `'\uD835'` and `'\uDD6B'`.

If you call `charAt(0)` on the string containing the single character $\mathbb{Z}$ (that is, the string `"\uD835\uDD6B"`), you only get the first half of a supplementary character.

Therefore, you should only use `char` values if you are absolutely sure that you won't need to encode supplementary characters.

RANDOM FACT 4.2

International Alphabets

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 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. German computer keyboards have keys for these characters (see Figure 5).

This poses a problem for computer users and designers. The American standard character encoding (called ASCII, for American Standard Code for Information Interchange) specifies 128 codes: 52 upper- and lowercase characters, 10 digits, 32 typographical symbols, and 34 control characters (such as space, newline, and 32 others for controlling printers and other devices). The umlaut and double-s are not among them. Some German data processing systems replace seldom-used ASCII characters with German letters: `[ \ ] { | } ~` are replaced with Ä Ö Ü ä ö ü ß. Most people can live without those ASCII characters, but programmers using Java definitely cannot. Other encoding schemes take advantage of the fact that one byte can encode 256 different characters, but only 128 are standardized by ASCII. Unfortunately, there are multiple incompatible standards for using the remaining 128 characters, resulting in a certain amount of aggravation among e-mail correspondents in different European countries.

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 (see Figure 6). To complicate matters, scripts like Hebrew and Arabic are written from right to left instead of from left to right, and many of these scripts have characters that stack above or below other characters, as those marked with a dotted circle in Figure 6 do in Thai. Each of these alphabets has between 30 and 100 letters, and the countries using them have established encoding standards for them.

The situation is much more dramatic in languages that use Chinese script: the Chinese dialects, Japanese, and Korean. The Chinese script is not alphabetic but ideographic—a character represents an idea or thing rather than a single sound. (See Figure 7; can you identify the characters for soup, chicken, and wonton?) Most words are made up of one, two, or three of these ideographic characters. Tens of thousands of ideographs are in active use, and China, Taiwan, Hong Kong, Japan, and Korea developed incompatible encoding standards for them.

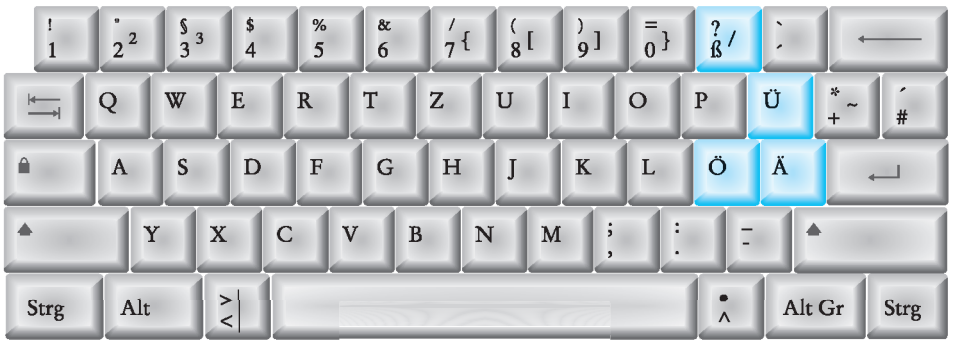


Figure 5 A German Keyboard

	จ	ฐ	ธ	ภ	ศ	ะ	ุ	เ	่	๐	๘		เ◌
ก	ฉ	ท	น	ม	ษ	ั	ู	แ	้	๑	๙		แ◌
ข	ช	ฒ	บ	ย	ส	า	◌	โ	๊	๒	ๆ		โ◌
ข	ช	ณ	ป	ร	ห	ำ		ใ	◌	๓	๛		ใ◌
ค	ฌ	ด	ผ	ฤ	ฬ	ิ		ไ	์	๔			ไ◌
ค	ญ	ต	ฝ	ล	อ	ี		ๅ	◌	๕			
ฆ	ฎ	ถ	พ	ฦ	ฮ	ึ		ๆ	๎	๖			
ง	ฏ	ท	ฟ	ว	ฯ	ื		็		๗			

Figure 6 The Thai Alphabet

The inconsistencies among character encodings have been a major nuisance for international electronic communication and for software manufacturers vying for a global market. Between 1988 and 1991 a consortium of hardware and software manufacturers developed a uniform encoding scheme called Unicode that is expressly designed to encode text in all written languages of the world (see reference [1]). In the first version of Unicode, about 39,000 characters were given codes, including 21,000 Chinese ideographs. A 2-byte code (which can encode over 65,000 characters) was chosen. It was thought to leave ample space for expansion for esoteric scripts, such as Egyptian hieroglyphs and the ancient script used on the island of Java.

Java was one of the first programming languages to embrace Unicode. The primitive type `char` denotes a 2-byte Unicode character. (All Unicode characters can be stored in Java strings, but which ones can actually be displayed depends on your computer system.)

CLASSIC SOUPS

			Sm.	Lg.
清燉雞湯	57.	House Chicken Soup (Chicken, Celery, Potato, Onion, Carrot)	1.50	2.75
雞飯湯	58.	Chicken Rice Soup	1.85	3.25
雞麵湯	59.	Chicken Noodle Soup	1.85	3.25
廣東雲吞	60.	Cantonese Wonton Soup	1.50	2.75
蕃茄蛋湯	61.	Tomato Clear Egg Drop Soup	1.65	2.95
雲吞湯	62.	Regular Wonton Soup	1.10	2.10
酸辣湯	63.	Hot & Sour Soup	1.10	2.10
蛋花湯	64.	Egg Drop Soup	1.10	2.10
雲蛋湯	65.	Egg Drop Wonton Mix	1.10	2.10
豆腐菜湯	66.	Tofu Vegetable Soup	NA	3.50
雞玉米湯	67.	Chicken Corn Cream Soup	NA	3.50
蟹肉玉米湯	68.	Crab Meat Corn Cream Soup	NA	3.50
海鮮湯	69.	Seafood Soup	NA	3.50

Figure 7 A Menu with Chinese Characters

Unfortunately, in 2003, the inevitable happened. Another large batch of Chinese ideographs had to be added to Unicode, pushing it beyond the 16-bit limit. Now, some characters need to be encoded with a pair of `char` values.

4.7 Reading Input

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

The Java programs that you have made so far have constructed objects, called methods, printed results, and exited. They were not interactive and took no user input. In this section, you will learn one method for reading user input.

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—it can only read one byte at a time. Finally, in Java version 5.0, a `Scanner` class was added that lets you read keyboard input in a convenient manner.

To construct a `Scanner` object, simply pass the `System.in` object to the `Scanner` constructor:

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

You can create a scanner out of any input stream (such as a file), but you will usually want to use a scanner to read keyboard input from `System.in`.

Once you have a scanner, you use the `nextInt` or `nextDouble` methods to read the next integer or floating-point number.

```
System.out.print("Enter quantity: ");
int quantity = in.nextInt();

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

When the `nextInt` or `nextDouble` method is called, the program waits until the user types a number and hits the Enter key. You should always provide instructions for the user (such as `"Enter quantity:"`) before calling a `Scanner` method. Such an instruction is called a *prompt*.

The `nextLine` method returns the next line of input (until the user hits the Enter key) as a `String` object. The `next` method returns the next *word*, terminated by any *white space*, that is, a space, the end of a line, or a tab.

```
System.out.print("Enter city: ");
String city = in.nextLine();

System.out.print("Enter state code: ");
String state = in.next();
```

Here, we use the `nextLine` method to read a city name that may consist of multiple words, such as `San Francisco`. We use the `next` method to read the state code (such as `CA`), which consists of a single word.

Here is an example of a test class that takes user input. This class tests the `CashRegister` class and allows the user to supply a purchase price and coin counts.

File InputTester.java

```
1  import java.util.Scanner;
2
3  /**
4     This class tests console input.
5  */
6  public class InputTester
7  {
8     public static void main(String[] args)
9     {
10       Scanner in = new Scanner(System.in);
11
12       CashRegister register = new CashRegister();
13
14       System.out.print("Enter price: ");
15       double price = in.nextDouble();
16       register.recordPurchase(price);
17
18       System.out.print("Enter dollars: ");
19       int dollars = in.nextInt();
20       System.out.print("Enter quarters: ");
21       int quarters = in.nextInt();
22       System.out.print("Enter dimes: ");
23       int dimes = in.nextInt();
24       System.out.print("Enter nickels: ");
25       int nickels = in.nextInt();
26       System.out.print("Enter pennies: ");
27       int pennies = in.nextInt();
28       register.enterPayment(dollars, quarters, dimes, nickels, pennies);
29
30       System.out.print("Your change is ");
31       System.out.println(register.giveChange());
32    }
33 }
```

Output

```
Enter price: 7.55
Enter dollars: 10
Enter quarters: 2
Enter dimes: 1
Enter nickels: 0
Enter pennies: 0
Your change is 3.05
```

Self Check

15. Why can't input be read directly from `System.in`?
16. Suppose `in` is a `Scanner` object that reads from `System.in`, and your program calls

```
String name = in.next();
```

What is the value of `name` if the user enters `John Q. Public`?

Advanced Topic 4.6

Formatting Numbers

The default format for printing numbers is not always what you would like. For example, consider the following code segment:

```
double total = 3.50;
final double TAX_RATE = 8.5; // Tax rate in percent
double tax = total * TAX_RATE / 100; // tax is 0.2975
System.out.println("Total: " + total);
System.out.println("Tax:   " + tax);
```

The output is

```
Total: 3.5
Tax:   0.2975
```

You may prefer the numbers to be printed with two digits after the decimal point, like this:

```
Total: 3.50
Tax:   0.30
```

You can achieve this with the `printf` method of the `PrintStream` class. (Recall that `System.out` is an instance of `PrintStream`.) The first parameter of the `printf` method is a *format string* that shows how the output should be formatted. The format string contains characters that are simply printed, and *format specifiers:* codes that start with a `%` character and end with a letter that indicates the format type. There are quite a few formats—Table 3 shows the most important ones. The remaining parameters of `printf` are the values to be formatted. For example,

```
System.out.printf("Total:%5.2f", total);
```

prints the string `Total:`, followed by a floating-point number with a *width* of 5 and a *precision* of 2. The width is the total number of characters to be printed: in our case, a space, the digit 3, a period, and two digits. If you increase the width, more spaces are added. The precision is the number of digits after the decimal point.

This simple use of `printf` is sufficient for most formatting needs. Once in a while, you may see a more complex example, such as this one:

```
System.out.printf("%-6s%5.2f%n", "Tax:", total);
```

Here, we have three format specifiers. The first one is `%-6s`. The `s` indicates a string. The hyphen is a *flag*, modifying the format. (See Table 4 for the most common format flags. The flags immediately follow the `%` character.) The hyphen indicates left alignment. If the string

Table 3 Format Types

Code	Type	Example
d	Decimal integer	123
x	Hexadecimal integer	7B
o	Octal integer	173
f	Fixed floating-point	12.30
e	Exponential floating-point	1.23e+1
g	General floating-point (exponential notation used for very large or very small values)	12.3
s	String	Tax:
n	Platform-independent line end	

to be formatted is shorter than the width, it is placed to the left, and spaces are added to the right. (The default is right alignment, with spaces added to the left.) Thus, `%-6s` denotes a left-aligned string of width 6.

You have already seen `%5.2f`: a floating-point number of width 5 and precision 2. The final specifier is `%n`, indicating a platform-independent line end. In Windows, lines need to be terminated by *two* characters: a carriage return `'\r'` and a newline `'\n'`. In other operating systems, a `'\n'` suffices. The `%n` format emits the appropriate line terminators.

Moreover, this call to `printf` has two parameters. You can supply any number of parameter values to the `printf` method. Of course, they must match the format specifiers in the format string.

Table 4 Format Flags

Flag	Meaning	Example
-	Left alignment	1.23 followed by spaces
0	Show leading zeroes	001.23
+	Show a plus sign for positive numbers	+1.23
(	Enclose negative numbers in parentheses	(1.23)
,	Show decimal separators	12,300
^	Convert letters to uppercase	1.23E+1

The `format` method of the `String` class is similar to the `printf` method. However, it returns a string instead of producing output. For example, the call

```
String message = String.format("Total:%5.2f", total);
```

sets the `message` variable to the string `"Total: 3.50"`.

ADVANCED TOPIC 4.7

Reading Input from a Dialog Box

Prior to Java version 5.0, it was not an easy matter to read input in a console window. The easiest method was to create a separate pop-up window for each input (see Figure 8). This note tells you how to do that, in case you need to work with an older version of Java.

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);
```

Finally, whenever you call `JOptionPane.showInputDialog` in your programs, you need to add a line

```
System.exit(0);
```

to the end of your `main` method. The `showInputDialog` method starts a user interface thread to handle user input. When the `main` method reaches the end, that thread is still running, and your program won't exit automatically. To force the program to exit, you need to call the `exit` method of the `System` class. The parameter of the `exit` method is the status code of the program. A code of `0` denotes successful completion; you can use nonzero status codes to denote various error conditions.

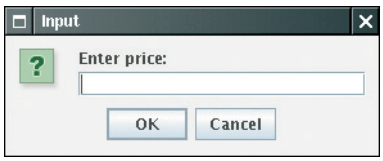


Figure 8 An Input Dialog Box

CHAPTER SUMMARY

1. Java has eight primitive types, including four integer types and two floating-point types.
2. A numeric computation overflows if the result falls outside the range for the number type.
3. Rounding errors occur when an exact conversion between numbers is not possible.
4. You use a cast (*typeName*) to convert a value to a different type.
5. Use the `Math.round` method to round a floating-point number to the nearest integer.
6. A `final` variable is a constant. Once its value has been set, it cannot be changed.
7. Use named constants to make your programs easier to read and maintain.
8. Assignment to a variable is not the same as mathematical equality.
9. The `++` and `--` operators increment and decrement a variable.
10. If both arguments of the `/` operator are integers, the result is an integer and the remainder is discarded.
11. The `%` operator computes the remainder of a division.
12. The `Math` class contains methods `sqrt` and `pow` to compute square roots and powers.
13. A static method does not operate on an object.
14. A string is a sequence of characters. Strings are objects of the `String` class.
15. Strings can be concatenated, that is, put end to end to yield a new longer string. String concatenation is denoted by the `+` operator.
16. Whenever one of the arguments of the `+` operator is a string, the other argument is converted to a string.
17. If a string contains the digits of a number, you use the `Integer.parseInt` or `Double.parseDouble` method to obtain the number value.
18. Use the `substring` method to extract a part of a string.
19. String positions are counted starting with `0`.
20. Use the `Scanner` class to read keyboard input in a console window.

FURTHER READING

1. `http://www.unicode.org/` The web site of the Unicode consortium. It contains character tables that show the Unicode values of characters from many scripts.

CLASSES, OBJECTS, AND METHODS INTRODUCED IN THIS CHAPTER

```
java.io.PrintStream
   printf
java.lang.Double
   parseDouble
java.lang.Integer
   parseInt
   toString
   MAX_VALUE
   MIN_VALUE
java.lang.Math
   E
   PI
   abs
   acos
   asin
   atan
   atan2
   ceil
   cos
   exp
   floor
   log
   max
   min
   pow
   round
   sin
   sqrt
   tan
   toDegrees
   toRadians
java.lang.String
   format
   substring
java.lang.System
   in
java.math.BigDecimal
   add
   multiply
   subtract
java.math.BigInteger
   add
   multiply
   subtract
java.util.Scanner
   next
   nextDouble
   nextInt
   nextLine
```

REVIEW EXERCISES

Exercise R4.1. Write the following mathematical expressions in Java.

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

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

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

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

Exercise R4.2. 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.
```
p = Math.atan2(z, Math.sqrt(x * x + y * y));
```

Exercise R4.3. What is wrong with this version of the quadratic formula?

```
x1 = (-b - Math.sqrt(b * b - 4 * a * c)) / 2 * a;
x2 = (-b + Math.sqrt(b * b - 4 * a * c)) / 2 * a;
```

Exercise R4.4. Give an example of integer overflow. Would the same example work correctly if you used floating-point? Give an example of a floating-point roundoff error. Would the same example work correctly if you used integers? For this exercise, you should assume that the values are represented in a sufficiently small unit, such as cents instead of dollars, so that the values don't have a fractional part.

Exercise R4.5. Write a test program that executes the following code:

```
CashRegister register = new CashRegister();
register.recordPurchase(19.93);
register.enterPayment(20, 0, 0, 0, 0);
System.out.print("Your change is ");
System.out.println(register.giveChange());
```

The program prints the total as `0.07000000000000028`. Explain why. Give a recommendation to improve the program so that users will not be confused.

Exercise R4.6. Let `n` be an integer and `x` a floating-point number. Explain the difference between

```
n = (int) x;
```

and

```
n = (int) Math.round(x);
```

Exercise R4.7. Let `n` be an integer and `x` a floating-point number. Explain the difference between

```
n = (int) (x + 0.5);
```

and

```
n = (int) Math.round(x);
```

For what values of `x` do they give the same result? For what values of `x` do they give different results?

Exercise R4.8. Explain the differences between `2`, `2.0`, `'2'`, `"2"`, and `"2.0"`.

Exercise R4.9. Explain what each of the following two program segments computes:

```
x = 2;
y = x + x;
```

and

```
s = "2";
t = s + s;
```

Exercise R4.10. Uninitialized variables can be a serious problem. Should you always initialize every variable with zero? Explain the advantages and disadvantages of such a strategy.

Exercise R4.11. True or false? (`x` is an `int` and `s` is a `String`)

a. `Integer.parseInt("" + x)` is the same as `x`

b. `"" + Integer.parseInt(s)` is the same as `s`

c. `s.substring(0, s.length())` is the same as `s`

Exercise R4.12. How do you get the first character of a string? The last character? How do you remove the first character? The last character?

Exercise R4.13. How do you get the last digit of an integer? The first digit? That is, if `n` is `23456`, how do you find out that the first digit is `2` and the last digit is `6`? Do not convert the number to a string. *Hint:* `%`, `Math.log`.

Exercise R4.14. This chapter contains several recommendations regarding variables and constants that make programs easier to read and maintain. Summarize these recommendations.

Exercise R4.15. What is a `final` variable? Can you define a `final` variable without supplying its value? (Try it out.)

Exercise R4.16. 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;
String s = "Hello";
String t = "World";
```

a. `x + n * y - (x + n) * y`

b. `m / n + m % n`

c. `5 * x - n / 5`

d. `Math.sqrt(Math.sqrt(n))`

e. `(int) Math.round(x)`

f. `(int) Math.round(x) + (int) Math.round(y)`

g. `s + t`

h. `s + n`

i. `1 - (1 - (1 - (1 - (1 - n))))`

j. `s.substring(1, 3)`

k. `s.length() + t.length()`

PROGRAMMING EXERCISES

Exercise P4.1. Enhance the `CashRegister` class by adding separate methods `enterDollars`, `enterQuarters`, `enterDimes`, `enterNickels`, and `enterPennies`. For example,

```
register.recordPurchase(20.37);
register.enterDollars(20);
register.enterQuarters(2);
System.out.println(register.giveChange()); // Prints 0.13
```

Exercise P4.2. Enhance the `CashRegister` class so that it keeps track of the total number of items in a sale. Count all recorded purchases and supply a method

```
int getItemCount()
```

that returns the number of items of the current purchase. Remember to reset the count at the end of the purchase.

Exercise P4.3. Write a program that prints the values

```
1
10
100
1000
10000
100000
1000000
10000000
100000000
1000000000
10000000000
100000000000
```

Implement a class

```
public class PowerGenerator
{
   /**
      Constructs a power generator.
      @param aFactor the number that will be multiplied by itself
   */
   public PowerGenerator(int aFactor) { . . . }

   /**
      Computes the next power.
   */
   public double nextPower() { . . . }
   . . .
}
```

Then supply a test class `PowerGeneratorTest` that calls `System.out.println(myGenerator.nextPower())` twelve times.

Exercise P4.4. Write a program that prompts the user for two numbers, 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)

To do so, implement a class

```
public class Pair
{
   /**
      Constructs a pair.
      @param aFirst the first value of the pair
      @param aSecond the second value of the pair
   */
   public Pair(double aFirst, double aSecond) { . . . }

   /**
      Computes the sum of the values of this pair.
      @return the sum of the first and second values
   */
   public double getSum() { . . . }
   . . .
}
```

Then implement a class `PairTest` that reads in two numbers, constructs a `Pair` object, invokes its methods, and prints the results.

Exercise P4.5. Write a program that reads in four integers and prints their sum and average. Define a class `DataSet` with methods

- `void addValue(int x)`
- `int getSum()`
- `double getAverage()`

Hint: Keep track of the sum and the count of the values. Then write a test program `DataSetTest` that reads four numbers and calls `addValue` four times.

Exercise P4.6. Write a program that reads in four integers and prints the largest and smallest value that the user entered. Use a class `DataSet` with methods

- `void addValue(int x)`
- `int getLargest()`
- `int getSmallest()`

Keep track of the smallest and largest value that you've seen so far. Then use the `Math.min` and `Math.max` methods to update it in the `addValue` method. What should you use as initial values? *Hint:* `Integer.MIN_VALUE`, `Integer.MAX_VALUE`.

Write a test program `DataSetTest` that reads four numbers and calls `addValue` four times.

Exercise P4.7. Write a program that prompts the user for a measurement in meters and then converts it into miles, feet, and inches. Use a class

```
public class Converter
{
   /**
      Constructs a converter that can convert between two units.
      @param aConversionFactor the factor with which to multiply
      to convert to the target unit
   */
   public Converter(double aConversionFactor) { . . . }

   /**
      Converts from a source measurement to a target measurement.
      @param fromMeasurement the measurement
      @return the input value converted to the target unit
   */
   public double convertTo(double fromMeasurement) { . . . }
}
```

Then construct three instances, similar to this example:

```
final double MILE_TO_KM = 1.609;
Converter milesToMeters = new Converter(1000 * MILE_TO_KM);
```

Exercise P4.8. Write a program that prompts the user for a radius and then prints

- The area and circumference of the circle with that radius
- The volume and surface area of the sphere with that radius

Define classes `Circle` and `Sphere`.

Exercise P4.9. Implement a class `SodaCan` whose constructor receives the height and diameter of the soda can. Supply methods `getVolume` and `getSurfaceArea`. Supply a `SodaCanTest` class that tests your class.

Exercise P4.10. Write a program that asks the user for the length of the sides of a square. Then print

- The area and perimeter of the square
- The length of the diagonal (use the Pythagorean theorem)

Define a class `Square`.

Exercise P4.11. *Giving change.* Enhance the `CashRegister` class so that it directs a cashier how to give change. The cash register computes the amount to be returned to the customer, in pennies.

Add the following methods to the `CashRegister` class:

- `int giveDollars()`
- `int giveQuarters()`
- `int giveDimes()`
- `int giveNickels()`
- `int givePennies()`

Each method computes the number of dollar bills or coins to return to the customer, and reduces the change due by the returned amount. You may assume that the methods are called in this order. For example,

```
CashRegister register = new CashRegister();
register.recordPurchase(8.37);
register.enterPayment(10, 0, 0, 0, 0);
double dollars = register.returnDollars(); // Returns 1
double quarters = register.returnQuarters(); // Returns 2
double dimes = register.returnDimes(); // Returns 1
double nickels = register.returnNickels(); // Returns 0
double pennies = register.returnPennies(); // Returns 3
```

Exercise P4.12. Write a program that reads in an integer and breaks it into a sequence of individual digits in reverse order. For example, the input 16384 is displayed as

```
4
8
3
6
1
```

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

Define a class `DigitExtractor`:

```
public class DigitExtractor
{
   /**
      Constructs a digit extractor that gets the digits
      of an integer in reverse order.
      @param anInteger the integer to break up into digits
   */
   public DigitExtractor(int anInteger) { . . . }

   /**
      Returns the next digit to be extracted.
      @return the next digit
   */
   public int nextDigit() { . . . }
}
```

Then call `System.out.println(myExtractor.nextDigit())` five times.

Exercise P4.13. Implement a class `QuadraticEquation` whose constructor receives the coefficients `a`, `b`, `c` of the quadratic equation $ax^2 + bx + c = 0$. Supply methods `getSolution1` and `getSolution2` that get the solutions, using the quadratic formula. Write a test class `QuadraticEquationTest` that prompts the user for the values of `a`, `b`, and `c`, constructs a `QuadraticEquation` object, and prints the two solutions.

Exercise P4.14. 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 time:

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

Implement a class `TimeInterval` whose constructor takes two military times. The class should have two methods `getHours` and `getMinutes`.

Exercise P4.15. *Writing large letters.* A large letter `H` can be produced like this:

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

Define a class `LetterH` with a method

```
String getLetter()
{
   return "*   *\n*   *\n*****\n*   *\n*   *\n";
}
```

Do the same for the letters `E`, `L`, and `O`. Then write the message

```
H
E
L
L
O
```

in large letters.

Exercise P4.16. Write a program that prints a Christmas tree:

```
   /\
  /  \
 /    \
/      \
--------
  "  "
  "  "
  "  "
```

Remember to use escape sequences.

Exercise P4.17. 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. Implement a class `Month` whose constructor parameter is the month number and whose `getName` method returns the month name.

Exercise P4.18. Write a program to compute the date of Easter Sunday. Easter Sunday is the first Sunday after the first full moon of spring. 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
b = 20
c = 1
d = 5, e = 0
g = 6
h = 18
j = 0, k = 1
m = 0
r = 6
n = 4
p = 15
```

Therefore, in 2001, Easter Sunday fell on April 15. Write a class `Easter` with methods `getEasterSundayMonth` and `getEasterSundayDay`.

PROGRAMMING PROJECTS

Project 4.1. 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

Of course, you should 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.

This is a good team project for two students. Both students should agree on the `Triangle` interface. One student implements the `Triangle` class, the other simultaneously implements the user interaction and formatting.

Project 4.2. 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.

ANSWERS TO SELF-CHECK QUESTIONS

1. `int` and `double`.
2. When the fractional part of `x` is ≥0.5.
3. By using a cast: `(int) Math.round(x)`.
4. The first definition is used inside a method, the second inside a class.
5. (1) You should use a named constant, not the "magic number" 3.14.
   (2) 3.14 is not an accurate representation of π.
6. The statement adds the `amount` value to the `balance` variable.
7. One less than it was before.
8. 17 and 29.
9. Only `s3` is divided by 3. To get the correct result, use parentheses. Moreover, if `s1`, `s2`, and `s3` are integers, you must divide by `3.0` to avoid integer division:
   ```
   (s1 + s2 + s3) / 3.0
   ```
10. $\sqrt{x^2 + y^2}$
11. `x` is a number, not an object, and you cannot invoke methods on numbers.
12. No—the `println` method is called on the object `System.out`.
13. `s` is set to the string `Agent5`.
14. The strings `"i"` and `"ssissi"`.
15. The class only has a method to read a single byte. It would be very tedious to form characters, strings, and numbers from those bytes.
16. The value is `"John"`. The `next` method reads the next *word*.