Chapter 3

Fundamental Data Types

CHAPTER GOALS

To understand integer and floating-point numbers

- To recognize the limitations of the `int` and `double` types and the overflow and roundoff errors that can result
- To write arithmetic expressions in Java
- To use the `String` type to define and manipulate character strings
- To learn about the `char` data type
- To learn how to read program input
- To understand the copy behavior of primitive types and object references

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.

CHAPTER CONTENTS

3.1 Number Types

In this chapter, we will use a `Purse` class to demonstrate several important concepts. We won't yet reveal the implementation of the purse, but here is the public interface:

```
public class Purse
{
   /**
      Constructs an empty purse.
   */
   public Purse()
   {
      // implementation
   }

   /**
      Add nickels to the purse.
      @param count the number of nickels to add
   */
   public void addNickels(int count)
   {
      // implementation
   }

   /**
      Add dimes to the purse.
      @param count the number of dimes to add
```

```
   */
   public void addDimes(int count)
   {
      // implementation
   }

   /**
      Add quarters to the purse.
      @param count the number of quarters to add
   */
   public void addQuarters(int count)
   {
      // implementation
   }

   /**
      Get the total value of the coins in the purse.
      @return the sum of all coin values
   */
   public double getTotal()
   {
      // implementation
   }

   // private instance variables
}
```

Read through the public interface and ask yourself whether you can figure out how to use `Purse` objects. You should find this straightforward. There is a constructor to make a new, empty purse:

```
Purse myPurse = new Purse();
```

You can add nickels, dimes, and quarters. (For simplicity, we don't bother with pennies, half dollars, or dollar coins.)

```
myPurse.addNickels(3);
myPurse.addDimes(1);
myPurse.addQuarters(2);
```

Now you can ask the purse object about the total value of the coins in the purse:

```
double totalValue = myPurse.getTotal(); // returns 0.75
```

The `int` type denotes integers: numbers without fractional parts.

If you look closely at the methods to add coins, you will see an unfamiliar data type. The `count` parameter has type `int`, which denotes an *integer* type. An integer is a number without a fractional part. For example, 3 is an integer, but 0.05 is not. The number zero and negative numbers are integers. Thus, the `int` type is more restrictive than the `double` type that you saw in Chapter 2.

Why have both integer and floating-point number types? Your calculator doesn't have a separate integer type. It uses floating-point numbers for all calculations. Why don't we just use the `double` type for the coin counts?

There are two reasons for having a separate integer type: one philosophical and one pragmatic. In terms of philosophy, when we think about real purses and modern American coins, we recognize that there can be only a whole number of nickels, say, in a purse. If we were to saw a nickel in half, the halves would be worthless, and dropping one of them into a purse would not increase the amount of money in the purse. By specifying that the number of nickels is an integer, we make that observation into an explicit assumption in our model. The program would have worked just as well with floating-point numbers to count the coins, but it is generally a good idea to choose programming solutions that document one's intentions. Pragmatically speaking, integers are more efficient than floating-point numbers. They take less storage space, are processed faster on some platforms, and don't cause rounding errors.

Now let's start implementing the `Purse` class. Any `Purse` object can be described by the number of nickels, dimes, and quarters that the purse currently contains. Thus, we use three instance variables to represent the state of a `Purse` object:

```
public class Purse
{
   . . .
   private int nickels;
   private int dimes;
   private int quarters;
}
```

Now we can implement the `getTotal` method simply:

```
public double getTotal()
{
   return nickels * 0.05 + dimes * 0.1 + quarters * 0.25;
}
```

In Java, multiplication is denoted by an asterisk `*`, not a raised dot · or a cross ×, because there are no keys for these symbols on most keyboards. For example, $d \cdot 10$ is written as `d * 10`. Do not write commas or spaces in numbers in Java. For example, 10,150.75 must be entered as `10150.75`. To write numbers in exponential notation in Java, use `E` *n* instead of "$\times 10^n$". For example, 5.0×10^{-3} is written as `5.0E-3`.

The `getTotal` method computes the value of the expression

```
nickels * 0.05 + dimes * 0.1 + quarters * 0.25
```

That value is a floating-point number, because multiplying an integer (such as `nickels`) by a floating-point number (such as `0.05`) yields a floating-point number. The `return` statement returns the computed value as the method result, and the method exits.

Quality Tip 3.1

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 shorter variable names in your Java programs as well:

```
public class Purse
{
   . . .
   private int n;
   private int d;
   private int q;
}
```

Compare this with the previous one, though. Which one is easier to read? There is no comparison. Just reading `nickels` is a lot less trouble than reading `n` and then *figuring out* that it must mean "nickels".

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 `n` must stand for nickels, 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 `n` means when you look at the code six months from now?

Of course, you could use comments:

```
public class Purse
{
   . . .
   private int n; // nickels
   private int d; // dimes
   private int q; // quarters
}
```

That makes the definitions pretty clear. But in the `getTotal` method, you'd still have a rather cryptic computation `n * 0.05 + d * 0.1 + q * 0.25`. Descriptive variable names are a better choice, because they make your code easy to read without requiring comments.

Advanced Topic 3.1

Numeric Ranges and Precisions

Unfortunately, `int` and `double` values do suffer from one problem: They cannot represent arbitrarily large integer or floating-point numbers. Integers have a range of −2,147,483,648 (about −2 billion) to +2,147,483,647 (about 2 billion). See Advanced Topic 3.4 for an explanation of these values. If you need to refer to these boundaries in your program, use the constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE`, which are defined in a class called `Integer`. If you want to represent the world population, you can't use an `int`. Double-precision floating-point numbers are somewhat less limited; they can go up to more than 10^{300}. However, `double` floating-point numbers suffer from a different problem: *precision*. They store only about 15 significant digits. Suppose your customers might find the price of three hundred trillion dollars ($300,000,000,000,000) for your product a bit excessive, so you want to reduce it by five cents to a more reasonable-looking $299,999,999,999,999.95. Try running the following program:

```
class AdvancedTopic3_1
{
```

```
      public static void main(String[] args)
      {
         double originalPrice = 3E14;
         double discountedPrice = originalPrice - 0.05;
         double discount = originalPrice - discountedPrice;
            // should be 0.05;
         System.out.println(discount);
            // prints 0.0625;
      }
   }
```

The program prints out `0.0625`, not `0.05`. It is off by more than a penny!

For most of the programming projects in this book, the limited range and precision of `int` and `double` are acceptable. Just bear in mind that overflows or loss of precision can occur.

Advanced Topic 3.2

Other Number Types

If `int` and `double` are not sufficient for your computational needs, there are other data types to which you can turn. When the range of integers is not sufficient, the simplest remedy is to use the `long` type. *Long integers* have a range from −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.

To specify a long integer constant, you need to append the letter `L` after the number value. For example,

```
long price = 300000000000000L;
```

There is also an integer type `short` with shorter-than-normal integers, having a range of −32,768 to 32,767. Finally, there is a type `byte` with a range of −128 to 127.

The `double` type can represent about 15 decimal digits. There is a second floating-point type, called `float`, whose values take half the storage space. Computations involving `float` execute a bit faster than those involving `double`, but the precision of `float` values—about 7 decimal digits—is insufficient for many programs. However, some graphics routines require you to use `float` values.

By the way, the name "floating-point" comes from the fact that the numbers are represented in the computer as a sequence of the significant digits and an indication of the position of the decimal point. For example, the numbers 250, 2.5, 0.25, and 0.025 all have the same decimal digits: 25. When a floating-point number is multiplied or divided by 10, only the position of the decimal point changes; it "floats". This representation corresponds to numbers written in "exponential" or "scientific" notation, such as 2.5×10^2. (Actually, internally the numbers are represented in base 2, as binary numbers, but the principle is the same. See Advanced Topic 3.4 for more information on binary numbers.) Sometimes `float` values are called "single-precision", and of course `double` values are "double-precision" floating-point 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 (`+ - * /`) with them. Instead, you have to use methods called `add`, `subtract`, `multiply`, and `divide`. Here is an example of how to create two big numbers and how to multiply them.

```
BigInteger a = new BigInteger("123456789");
BigInteger b = new BigInteger("987654321");
BigInteger c = a.multiply(b);
System.out.println(c); // prints 121932631112635269
```

Random Fact 3.1

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 huge 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 has 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 chance and pocketed the money?

3.2 Assignment

Here is the constructor of the `Purse` class:

```
public Purse()
{
   nickels = 0;
   dimes = 0;
   quarters = 0;
}
```

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, in the `addNickels` method.

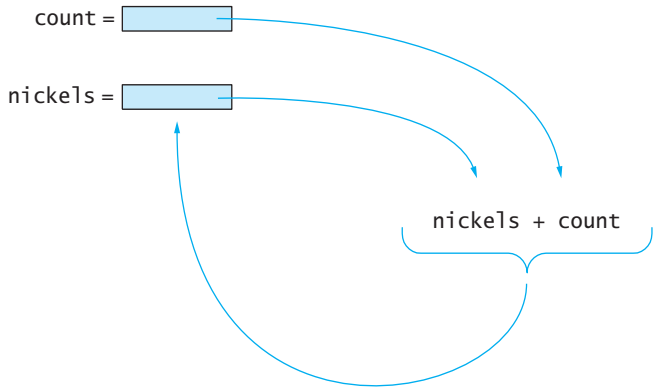


Figure 1

Assignment

```
public void addNickels(int count)
{
   nickels = nickels + count;
}
```

It means, "Compute the value of the expression `nickels + count`, *and place the result again into the variable* `nickels`." (See Figure 1.)

The = sign doesn't mean that the left-hand side is *equal* to the right-hand side but that the right-hand-side value is copied into the left-hand-side variable. You should not confuse this *assignment operation* with the = 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. For example, in Java it is perfectly legal to write

```
nickels = nickels + 1;
```

It means to look up the value stored in the variable `nickels`, to add 1 to it, and to stuff the sum back into `nickels`. (See Figure 2.) The net effect of executing this

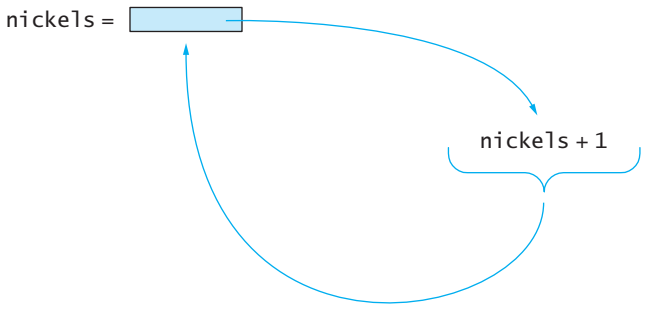


Figure 2

Incrementing a Variable

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

statement is to increment `nickels` by 1. Of course, in mathematics it would make no sense to write that $n = n + 1$; no integer can equal itself plus 1.

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.

Consider once more the statement `nickels = nickels + 1`. This statement increments the `nickels` variable. For example, if `nickels` was 3 before execution of the statement, it is set to 4 afterwards. This increment operation is so common when writing programs that there is a special shorthand for it, namely

```
nickels++;
```

This statement has exactly the same effect—namely, to add 1 to `nickels`—but it is easier to type. As you might have guessed, there is also a decrement operator `--`. The statement

```
nickels--;
```

subtracts 1 from `nickels`.

Advanced Topic 3.3

Combining Assignment and Arithmetic

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

```
nickels += count;
```

is a shortcut for

```
nickels = nickels + count;
```

Similarly,

```
nickels *= 2;
```

is another way of writing

```
nickels = nickels * 2;
```

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

Productivity Hint 3.1

Avoid Unstable Layout

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

However, you should 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. Suppose you have a comment that goes over multiple lines:

```
// In this test class, we compute the value of a set of coins.
// We add a number of nickels, dimes, and quarters
// to a purse. We then get and display the total value.
```

When the program is extended to work for half-dollar coins as well, you must modify the comment to reflect that change.

```
// In this test class, we compute the value of a set of coins.
// We add a number of nickels, dimes, quarters, and
half-dollars // to a purse. We then get and display the total
value.
```

Now you need to rearrange the // to fix up the comment. This scheme is a *disincentive* to keep comments up to date. Don't do it. Instead, for comments that are longer than one line, use the /*...*/ style for comments, and block off the entire comment like this:

```
/*
   In this test class, we compute the value of a set of coins.
   We add a number of nickels, dimes, and quarters
   to a purse. We then get and display the total value.
*/
```

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. As a consequence, you should avoid layout schemes that are hard to maintain.

3.3 Constants

Consider once again the `getTotal` method, paying attention to whether it is easy to understand the code.

```
public double getTotal()
{
   return nickels * 0.05 + dimes * 0.1 + quarters * 0.25;
}
```

Most of the code is self-documenting. However, the three numeric quantities, 0.05, 0.1, and 0.25, 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 nickelValue = 0.05;
double dimeValue = 0.1;
double quarterValue = 0.25;
return nickels * nickelValue
   + dimes * dimeValue
   + quarters * quarterValue;
```

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 more coins are added to the purse. But `nickelValue` is *always* 0.05. It is a *constant*. 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 the improved version of the `getTotal` method:

```
public double getTotal()
{
   final double NICKEL_VALUE = 0.05;
   final double DIME_VALUE = 0.1;
   final double QUARTER_VALUE = 0.25;
   return nickels * NICKEL_VALUE
      + dimes * DIME_VALUE
      + quarters * QUARTER_VALUE;
}
```

In this example, the constants are needed only inside one method of the class. Frequently, a constant value is needed in several methods. Then you need to declare it together with the instance variables of the class and tag it as `static final`. The meaning of the keyword `static` will be explained in Chapter 6.

```
public class Purse
{
   // methods
   . . .

   // constants
   private static final double NICKEL_VALUE = 0.05;
   private static final double DIME_VALUE = 0.1;
   private static final double QUARTER_VALUE = 0.25;

   // instance variables
   private int nickels;
   private int dimes;
   private int quarters;
}
```

Here we defined the constants to be `private` because we didn't think they were of interest to users of the `Purse` class. However, it is also possible to declare constants as `public`:

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

Then methods of other classes can access the constant as `Purse.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;
```

Syntax 3.1 : 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 of a particular type

File Purse.java

```
/**
   A purse computes the total value of a collection of coins.
*/
public class Purse
{
   /**
      Constructs an empty purse.
   */
   public Purse()
   {
      nickels = 0;
      dimes = 0;
      quarters = 0;
   }

   /**
      Add nickels to the purse.
      @param count the number of nickels to add
   */
   public void addNickels(int count)
   {
      nickels = nickels + count;
   }
   /**
      Add dimes to the purse.
      @param count the number of dimes to add
```

```
27     */
28     public void addDimes(int count)
29     {
30        dimes = dimes + count;
31     }
32
33     /**
34        Add quarters to the purse.
35        @param count the number of quarters to add
36     */
37     public void addQuarters(int count)
38     {
39        quarters = quarters + count;
40     }
41
42     /**
43        Get the total value of the coins in the purse.
44        @return the sum of all coin values
45     */
46     public double getTotal()
47     {
48        return nickels * NICKEL_VALUE
49           + dimes * DIME_VALUE + quarters * QUARTER_VALUE;
50     }
51
52     private static final double NICKEL_VALUE = 0.05;
53     private static final double DIME_VALUE = 0.1;
54     private static final double QUARTER_VALUE = 0.25;
55
56     private int nickels;
57     private int dimes;
58     private int quarters;
59  }
```

File PurseTest.java

```
1  /**
2     This program tests the Purse class.
3  */
4  public class PurseTest
5  {
6     public static void main(String[] args)
7     {
8        Purse myPurse = new Purse();
9
10       myPurse.addNickels(3);
11       myPurse.addDimes(1);
12       myPurse.addQuarters(2);
13       double totalValue = myPurse.getTotal();
```

```
14         System.out.print("The total is ");
15         System.out.println(totalValue);
16      }
17   }
```

Quality Tip 3.2

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.

You should use a named constant instead:

```
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.

3.4 Arithmetic 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 further 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 3.1.

If you are interested only in the remainder of an integer division, use the % operator:

```
7 % 4
```

The % operator computes the remainder of a division.

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 the value of the coins in a purse in 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 = nickels * PENNIES_PER_NICKEL
   + dimes * PENNIES_PER_DIME
   + quarters * PENNIES_PER_QUARTER;

// 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.

To take the square root of a number, you use the Math.sqrt method. For example, $\sqrt{x}$ is written as Math.sqrt(x). To compute x^n, you write Math.pow(x, n). However, to compute x^2 it is significantly more efficient simply to compute x * x.

Common Error 3.1

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 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;
```

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

As you can see, the visual effect of the /, `Math.sqrt`, and `Math.pow` notations is to flatten out mathematical terms. 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

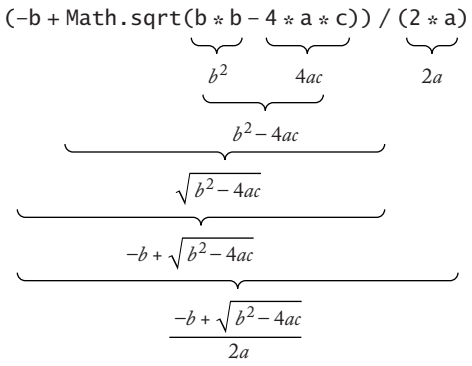
```
(-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a)
```

Figure 3 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 3.2.

Table 1 shows additional methods of the `Math` class. Inputs and outputs are floating-point numbers.

Figure 3

Analyzing an Expression



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 ($\tan^{-1}(y/x) \in [-\pi/2,\pi/2]$, 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 $\lvert x \rvert$

Table 1

Mathematical Methods

Common Error 3.2

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; and 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.

Productivity Hint 3.2

On-Line Help

The Java library has hundreds of classes and thousands of methods. It is neither necessary nor useful trying to memorize them. Instead, you should become familiar with using the on-line documentation. You can download the documentation from http://java.sun.com/j2se/1.3/docs.html. Install the documentation set and point your browser to your Java installation directory `/docs/api/index.html`. Alternatively, you can browse http://java.sun.com/j2se/1.3/docs/api/index.html. For example, if you are not sure how the `pow` method works, or cannot remember whether it was called `pow` or `power`, the on-line help

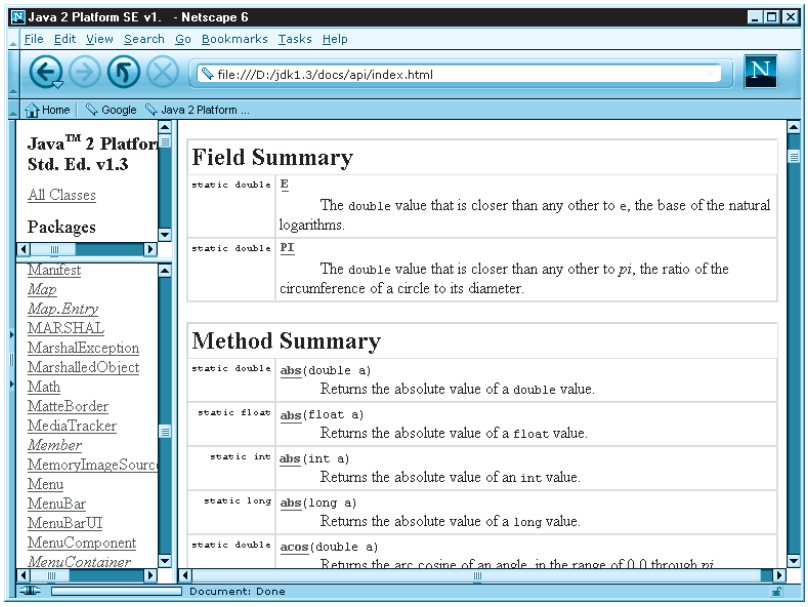


Figure 4

On-Line Help

can give you the answer quickly. Click on the `Math` class in the class window on the left, and look at the method summary in the main window (see Figure 4).

If you use `javadoc` to document your own classes, then this documentation format will look extremely familiar to you. The programmers who implement the Java library use `javadoc` themselves. They too document every class, every method, every parameter, and every return value, and then use `javadoc` to extract the documentation in HTML format.

Quality Tip 3.3

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 3.4

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 $\pm$ 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.

3.5 Calling Static Methods

In the preceding section, you encountered the `Math` class, which contains a collection of helpful methods for carrying out mathematical computations.

There is one important difference between the methods of the `Math` class, such as the `sqrt` method, and the methods that you have seen so far (such as `getTotal` or

println). The getTotal and println methods, as you have seen, operate on an object such as myPurse or System.out. In contrast, the sqrt method does not operate on any object. That is, you don't call

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

A static method does not operate on an object.

The reason is that, 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).

This call makes it appear as if the sqrt method is applied to an object called Math, because Math precedes sqrt just as myPurse precedes getTotal in a method call myPurse.getTotal(). 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 C and C++ that 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 an uppercase letter, you will likely confuse your fellow programmers. Therefore, we strongly recommend that you follow the standard naming convention.

Syntax 3.2 : 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

3.6 Type Conversion

When you make an assignment of an expression into a variable, the *types* of the variable and the expression must be compatible. For example, it is an error to assign

```
double total = "a lot"; // Error
```

because `total` is a floating-point variable and `"a lot"` is a string. It is, however, legal to store an integer expression in a `double` variable:

```
int dollars = 2;
double total = dollars; // OK
```

In Java, you cannot assign a floating-point expression to an integer variable.

```
double total = . . .;
int dollars = total; // Error
```

You must convert the floating-point value to integer with a *cast*:

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

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

The cast `(int)` converts the floating-point value `total` to an integer. The effect of the cast is to discard the fractional part. For example, if `total` is 13.75, then `dollars` is set to 13. If you want to convert the value of a floating-point expression to an integer, you need to enclose the expression in parentheses to ensure that it is computed first:

```
int pennies = (int)(total * 100);
```

This is different from the expression

```
int pennies = (int)total * 100;
```

In the second expression, `total` is *first* converted to an integer, and then the resulting integer is multiplied by 100. For example, if `total` is 13.75, then the first expression computes `total * 100`, or 1375, and then converts that value to the integer 1375. In the second expression, `total` is first cast to the integer 13, and then the integer is multiplied by 100, yielding 1300. Normally, you will want to apply the integer conversion *after* all other computations, so that your computations use the full precision of their input values. That means you should enclose your computation in parentheses and apply the cast to the expression in parentheses.

There is a good reason why you must use a cast in Java when you convert a floating-point number to an integer: The conversion *loses information*. You must confirm that you agree to that information loss. Java is quite strict about this. You must use a cast whenever there is the possibility of information loss. A cast always has the form `(`*typeName*`)`, for example `(int)` or `(float)`.

Actually, simply using an `(int)` cast to convert a floating-point number to an integer is not always a good idea. Consider the following example:

```
double price = 44.95;
int dollars = (int)price; // sets dollars to 44
```

What did you want to achieve? Did you want to get the number of dollars in the price? Then dropping the fractional part is the right thing to do. Or did you want to get the approximate dollar amount? Then you really want to *round up* when the fractional part is 0.5 or larger.

One way to round to the nearest integer is to add 0.5, then cast to an integer:

```
double price = 44.95;
int dollars = (int)(price + 0.5); // OK for positive values
System.out.print("The price is approximately $")
System.out.println(dollars); // prints 45
```

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

Adding 0.5 and casting to the `int` type works, because it turns all values that are between 44.50 and 45.4999... into 45.

Actually, there is a better way. Simply adding 0.5 works fine for positive numbers, but it doesn't work correctly for negative numbers. Instead, use the `Math.round` method in the standard Java library. It works for both positive and negative numbers. However, that method returns a `long` integer, because large floating-point numbers cannot be stored in an `int`. You need to cast the return value to an `int`:

```
int dollars = (int)Math.round(price); // better
```

Syntax 3.3 : Cast

(*typeName*) *expression*

Example:

```
(int)(x + 0.5)
(int)Math.round(100 * f)
```

Purpose:

To convert an expression to a different type

HOWTO 3.1

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 34 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 that the machine returns
- The number of penny stamps that the machine returns

Step 2 Work out examples by hand

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

Let's assume that a first-class stamp costs 34 cents and the customer inserts $1.00. That's enough for two stamps (68 cents) but not enough for three stamps ($1.02). Therefore, the machine returns 2 first-class stamps and 32 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/$0.34 ≈ 2.9412.

How do you get "2 stamps" out of 2.9412? It's the integer part. By discarding the fractional part, you get the number of whole stamps that 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 $0.32. It is the difference between $1.00 and 2 · $0.34. 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 $0.32 is left, the customer gets 32 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

HOWTO 2.1 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;
   }

   private double money;
   private static final double FIRST_CLASS_STAMP_PRICE =
      0.34;
}
```

Step 6 Test your class

Run a test program (or use 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.println("First class stamps: " +
   machine.giveFirstClassStamps());
System.out.println("Penny stamps: " +
   machine.givePennyStamps());
```

Check that the result is

```
First class stamps: 2
Penny stamps: 32
```

Common Error 3.3

Roundoff Errors

Roundoff errors are a fact of life when calculating with floating-point numbers. You probably have encountered that 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 3.4). 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:

```
int n = (int)Math.round(100 * f);
```

Note that the wrong result of the first computation is *not* caused by lack of precision. The problem lies with the wrong choice of rounding method. Dropping the fractional part, no matter how close it may be to 1, is not a good rounding method.

Advanced Topic 3.4

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 would be 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 ÷ 2 = 50 remainder 0
50 ÷ 2 = 25 remainder 0
25 ÷ 2 = 12 remainder 1
12 ÷ 2 = 6 remainder 0

$6 \div 2 = 3$ remainder 0
$3 \div 2 = 1$ remainder 1
$1 \div 2 = 0$ 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 in Common Error 3.3. 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 them all when converting the result to an integer.

3.7 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 basic types `int` and `double` start with a lowercase letter.)

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 `""`.

You already saw in Chapter 2 how 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. The concatenation operator in Java is very powerful. If *one of the expressions*, either to the left or the right of a + operator, is a string, then 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.

Since `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);
```

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

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
```

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
```

The `toUpperCase` and `toLowerCase` methods make strings with only upper- or lowercase characters. For example,

```
String greeting = "Hello";
System.out.println(greeting.toUpperCase());
System.out.println(greeting.toLowerCase());
```

This code segment prints `HELLO` and `hello`. Note that the `toUpperCase` and `toLowerCase` methods do not change the original `String` object `greeting`. They return new `String` objects that contain the uppercased and lowercased versions of the original string. In fact, *no* `String` methods modify the string object on which they operate. For that reason, strings are called *immutable* objects.

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

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

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

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

```
String greeting = "Hello, World!";
String sub = greeting.substring(0, 4);
   // sub is "Hell"
```

String positions are counted starting with 0.

The `substring` operation makes a string that consists of four characters taken from the string `greeting`. A curious aspect of the `substring` operation is the numbering of the starting and ending positions. Starting position 0 means "start at the beginning of the string". For technical reasons that used to be important but are no longer relevant, Java string position numbers start at 0. The first string position is labeled 0, the second one 1, and so on. For example, Figure 5 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.

Figure 5
String Positions

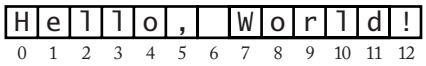
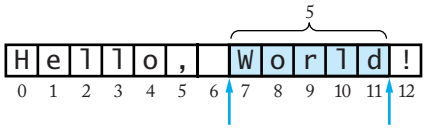


Figure 6
Extracting a Substring



Let us figure out how to extract the substring `"World"`. Count characters starting at 0, not 1. You find that `W`, the 8th character, has position number 7. The first character that you *don't* want, `!`, is the character at position 12 (see Figure 6). Therefore, the appropriate substring command is

```
String w = 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 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!"`.

Advanced Topic 3.5

Formatting Numbers

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

```
int quarters = 2;
int dollars = 3;
double total = dollars + quarters * 0.25; // price is 3.5
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 `NumberFormat` class in the `java.text` package. First, you must use the static method `getNumberInstance` to obtain a `NumberFormat` object. Then you set the maximum number of fraction digits to 2:

```
NumberFormat formatter =
   NumberFormat.getNumberInstance();
formatter.setMaximumFractionDigits(2);
```

Then the numbers are rounded to two digits. For example, 0.2875 will be converted to the string `"0.29"`. On the other hand, 0.2975 will be converted to `"0.3"`, not `"0.30"`. If you want trailing zeroes, you *also* have to set the minimum number of fraction digits to 2:

```
formatter.setMinimumFractionDigits(2);
```

Then you use the `format` method of that object. The result is a string that you can print.

```
formatter.format(tax)
```

returns the string `"0.30"`. The statement

```
System.out.println("Tax:   $" + formatter.format(tax));
```

rounds the value of `tax` to two digits after the decimal point and prints: `Tax: $0.30`.

The "number instance" formatter is useful because it lets you print numbers with as many fraction digits as desired. If you just want to print a currency value, the `getCurrencyInstance` method of the `NumberFormat` class produces a more convenient formatter. The "currency instance" formatter generates currency value strings, with the local currency symbol (such as `$` in the United States) and the appropriate number of digits after the decimal point (for example, two digits in the United States).

```
NumberFormat formatter = NumberFormat.getCurrencyInstance();
System.out.print(formatter.format(tax));
   // prints "$0.30"
```

3.8 Reading Input

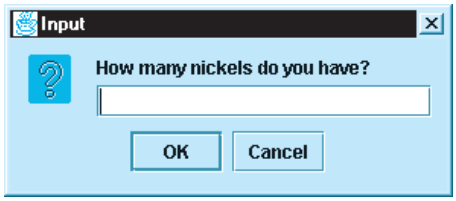
The Java programs that you have constructed so far have constructed objects, called methods, printed results, and exited. They were not interactive and took no user input. In this section, you will learn one method for reading user input.

The `JOptionPane.showInputDialog` method prompts the user for an input string.

The `JOptionPane` class has a static method `showInputDialog` that displays an input dialog (see Figure 7). The user can type any string into the input field and click the "OK" button. Then the `showInputDialog` method returns the string that the user entered. You should capture the user input in a string variable. For example,

```
String input =
   JOptionPane.showInputDialog("How many nickels do you have?");
```

Figure 7
An Input Dialog



Often you want the input as a number, not a string. Use the `Integer.parseInt` and `Double.parseDouble` methods to convert the string to a number:

```
int count = Integer.parseInt(input);
```

If the user doesn't type in a number, then the `parseInt` method *throws an exception*. An exception is a way for a method to indicate an error condition. You will see in Chapter 15 how to handle exceptions. Until then, we will simply rely on the default mechanism for exception handling. That mechanism terminates the program with an error message.

```
Exception in thread "main"
java.lang.NumberFormatException: x
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at InputTest.main(InputTest.java:10)
```

That doesn't make your programs very user-friendly. You will simply have to wait until Chapter 15 to make your programs bulletproof. In the meantime, you should assume that the user is cooperative and types in an actual number when you prompt for one. Since the users of your first programs are likely to be just yourself, your instructor, and your grader, that should not be a problem.

You must call `System.exit(0)` to exit a program that has a graphical user interface.

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. (See Chapter 20 for more information on threads.) 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.

Here is an example of a test class that takes user input. This class tests the `Purse` class and lets the user supply the numbers of nickels, dimes, and quarters.

File InputTest.java

```
1  import javax.swing.JOptionPane;
2
3  /**
4     This program tests input from an input dialog.
```

```
 5 */
 6 public class InputTest
 7 {
 8    public static void main(String[] args)
 9    {
10       Purse myPurse = new Purse();
11
12       String input = JOptionPane.showInputDialog(
13          "How many nickels do you have?");
14       int count = Integer.parseInt(input);
15       myPurse.addNickels(count);
16
17       input = JOptionPane.showInputDialog(
18          "How many dimes do you have?");
19       count = Integer.parseInt(input);
20       myPurse.addDimes(count);
21
22       input = JOptionPane.showInputDialog(
23          "How many quarters do you have?");
24       count = Integer.parseInt(input);
25       myPurse.addQuarters(count);
26
27       double totalValue = myPurse.getTotal();
28       System.out.println("The total is " + totalValue);
29
30       System.exit(0);
31    }
32 }
```

Admittedly, the program is not very elegant. It pops up three dialog boxes to collect input and then displays the output in the console window. You will learn in Chapter 12 how to write programs with more sophisticated graphical user interfaces.

Productivity Hint 3.3

Reading Exception Reports

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

```
Exception in thread "main" java.lang.NumberFormatException: x
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at InputTest.main(InputTest.java:10)
```

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 `NumberFormatException`
2. The line number of the code that contained the statement that caused the exception, such as `InputTest.java:10`

The name of the exception is always in the first line of the report, and it ends in `Exception`. If you get a `NumberFormatException`, then there was a problem with the format of some number. 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 lines that refer to

```
java.lang.Integer.parseInt(Unknown Source).
```

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 makes it completely obvious what went wrong, and you can easily fix your error.

Advanced Topic 3.6

Reading Console Input

You just saw how to read input from an input dialog. Admittedly, it is a bit strange to have dialogs pop up for every input. Some programmers prefer to read the input from the console window. Console input has one great advantage. As you will see in Chapter 6, you can put all your input strings into a file and redirect the console input to read from a file. That's a great help for program testing. However, console input is somewhat cumbersome to program. This note explains the details.

Console input reads from the `System.in` object. However, unlike `System.out`, which was ready-made for printing numbers and strings, `System.in` can only read *bytes*. Keyboard input consists of characters. To get a reader for characters, you have to turn `System.in` into an `InputStreamReader` object, like this:

```
InputStreamReader reader =
   newInputStreamReader(System.in);
```

Wrap `System.in` inside a `BufferedReader` to read input from the console window.

An input stream reader can read characters, but it can't read a whole string at a time. That makes it pretty inconvenient—you wouldn't want to piece together every input line from its individual characters. To overcome this limitation, you can turn an input stream reader into a `BufferedReader` object:

```
BufferedReader console =
   new BufferedReader(reader);
```

If you like, you can combine the two constructors:

```
BufferedReader console = new BufferedReader(
   new InputStreamReader(System.in));
```

Now you use the `readLine` method to read an input line, like this:

```
System.out.println(
   "How many nickels do you have?");
String input = console.readLine();
int count = Integer.parseInt(input);
```

When calling the `readLine` method of the `BufferedReader` class, you must tag the calling methods with `throws IOException`.

There is one remaining problem. When there is a problem with reading input, the `readLine` method generates an exception, just as the `parseInt` method does when you give it a string that isn't an integer. However, the `readLine` method generates an `IOException`, which is a *checked exception*, a more severe kind of exception than the `NumberFormatException` that the `parseInt` method generates. The Java compiler insists that you take one of two steps when you call a method that can throw a checked exception.

1. Handle the exception. You will see how to do that in Chapter 15
2. Acknowledge that you are not handling the exception. Then you have to indicate that your method can cause a checked exception because it calls another method that can cause that exception. You do that by tagging your method with a `throws` specifier, like this:

```
public static void main(String[] args) throws IOException
```

or

```
public void readInput(BufferedReader reader) throws IOException
```

There is no shame associated with acknowledging that your method might throw a checked exception—it is just "truth in advertising". Of course, in a professional program, you do need to handle all exceptions somewhere, and the `main` method won't throw any exceptions. You'll have to wait for Chapter 15 for the details.

Following this note is another version of the test program for the `Purse` class, this time reading input from the console. Figure 8 shows a typical program run.

File ConsoleInputTest.java

```
1 import java.io.BufferedReader;
2 import java.io.InputStreamReader;
3 import java.io.IOException;
4
5 /**
6    This program tests input from a console window.
7 */
```

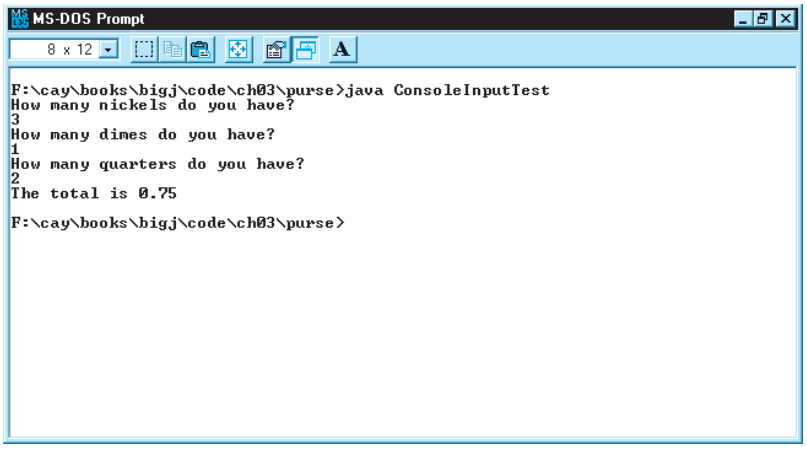


Figure 8

Reading Input from the Console

```
 8 public class ConsoleInputTest
 9 {
10    public static void main(String[] args) throws IOException
11    {
12       Purse myPurse = new Purse();
13
14
15       BufferedReader console = new BufferedReader(
16          new InputStreamReader(System.in));
17       System.out.println(
18          "How many nickels do you have?");
19       String input = console.readLine();
20       int count = Integer.parseInt(input);
21       myPurse.addNickels(count);
22
23       System.out.println("How many dimes do you have?");
24       input = console.readLine();
25       count = Integer.parseInt(input);
26       myPurse.addDimes(count);
27
28       System.out.println(
29          "How many quarters do you have?");
30       input = console.readLine();
31       count = Integer.parseInt(input);
32       myPurse.addQuarters(count);
33
34       double totalValue = myPurse.getTotal();
35       System.out.println("The total is " + totalValue);
```

```
36
37          System.exit(0);
38       }
39    }
```

3.9 Characters

A `char` value denotes a single character. Character constants are enclosed in single quotes.

Strings are composed of individual characters. Characters are values of the `char` type. A variable of type `char` can hold a single character.

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 1.1) 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 A6.

Characters have numeric values. For example, if you look at Appendix A6, you can see that the character `'H'` is actually encoded as the number 72.

The `charAt` method of the `String` class returns a character 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 character `'H'`.

Random Fact 3.2

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 a few times. German computer keyboards have keys for these characters (see Figure 9).

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.

Figure 9

German Keyboard

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, such as those used by the Windows and Macintosh operating systems, resulting in a certain amount of aggravation among European computer users and their American email correspondents.

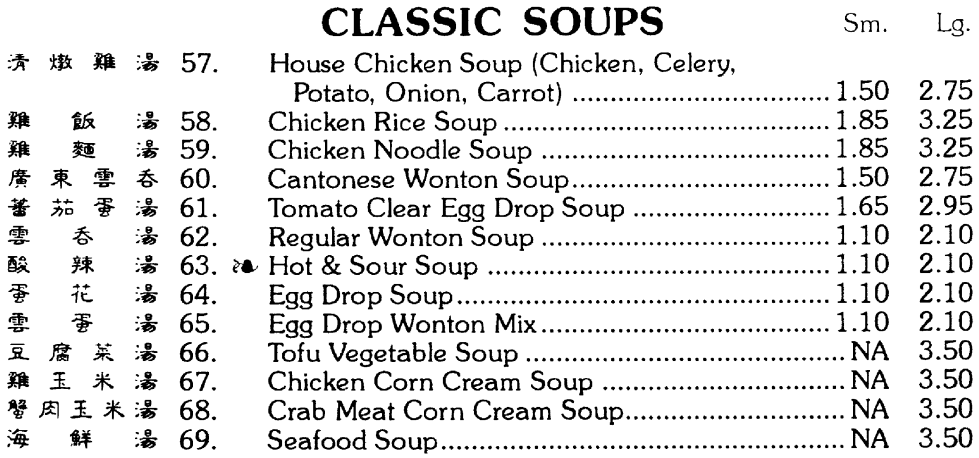
Many countries don't use the Roman script at all. Russian, Greek, Hebrew, Arabic, and Thai letters, to name just a few, have completely different shapes (see Figure 10). To complicate matters, scripts like Hebrew and Arabic are written from right to left instead of from left to right, and many of these scripts have characters that stack above or below other characters, as those marked with a dotted circle in Figure 10 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 the 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 11; can you identify the characters for soup, chicken, and wonton?) Most words are made up of one, two, or three of these ideographic characters. Over 50,000 ideographs are known, of which about 20,000 are in active use. Therefore, two bytes are needed to encode them. China, Taiwan, Japan, and Korea have incompatible encoding standards for them. (Japanese and Korean writing use a mixture of native syllabic and Chinese ideographic characters.)

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 16-bit encoding scheme called *Unicode* that is capable of encoding text in essentially all written languages of the world (see reference [1]). About 39,000 characters have been given codes, including 21,000 Chinese ideographs. A 16-bit code

Figure 10

The Thai Alphabet



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 11

A Menu with Chinese Characters

can incorporate 65,000 codes, so there is ample space for expansion. Future versions of the standard will be able to encode such scripts as Egyptian hieroglyphs and the ancient script used on the island of Java.

All Unicode characters can be stored in Java strings, but which ones can actually be displayed depends on your computer system.

3.10 Comparing Primitive Types and Objects

Number variables hold values. Object variables hold references.

In Java, every value is either a primitive type or an object reference. Primitive types are numbers (such as `int`, `double`, `char`, and the other number types listed in Advanced Topic 3.2) and the `boolean` type that you will encounter in Chapter 5. There is an important difference between primitive types and objects in Java. Primitive type variables hold values, but object variables don't hold objects—they hold references to objects. You can see the difference when you make a copy of a variable. When you copy a primitive type value, the original and the copy of the number are independent values. But when you copy an object reference, both the original and the copy are references to the same object.

A copy of an object reference is another reference to the same object.

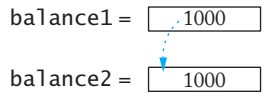
Consider the following code, which copies a number and then adds an amount to the copy (see Figure 12):

```
double balance1 = 1000;
double balance2 = balance1; // see Figure 12
balance2 = balance2 + 500;
```

Now the variable `balance1` contains the value 1000, and `balance2` contains 1500.

Figure 12

Copying Numbers



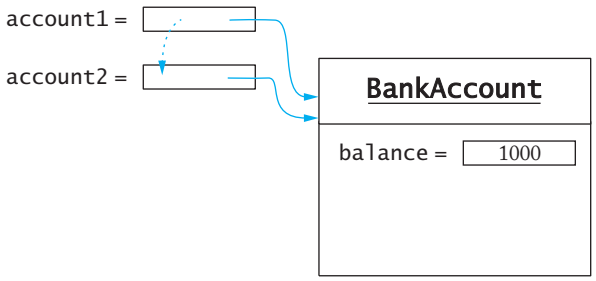


Figure 13

Copying Object References

Now consider the seemingly analogous code with `BankAccount` objects.

```
BankAccount account1 = new BankAccount(1000);
BankAccount account2 = account1; // see Figure 13
account2.deposit(500);
```

Unlike the preceding code, now *both* `account1` and `account2` have a balance of $1500.

What can you do if you actually need to make a true copy of an object—that is, a new object whose state is identical to an existing object? As you will see in Chapter 13, you can define a `clone` method for your classes to make such a copy. But in the meantime, you will simply have to construct a new object:

```
BankAccount account2 = new
   BankAccount(account1.getBalance());
```

Strings are objects; therefore, if you copy a `String` variable, you get two references to the same string object. However, unlike bank accounts, strings are *immutable*. None of the methods of the `String` class change the state of a `String` object. Thus, there is no problem in sharing string references.

Chapter Summary

1. The `int` type denotes integers, numbers without fractional part.
2. Assignment to a variable is not the same as mathematical equality.
3. The `++` and `--` operators increment and decrement a variable.
4. A `final` variable is a constant. Once its value has been set, it cannot be changed.

5. Use named constants to make your programs easier to read and maintain.
6. If both arguments of the `/` operator are integers, the result is an integer and the remainder is discarded.
7. The `%` operator computes the remainder of a division.
8. The `Math` class contains methods `sqrt` and `pow` to compute square roots and powers.
9. A static method does not operate on an object.
10. You use a cast `(`*typeName*`)` to convert a value to a different type.
11. Use the `Math.round` method to round a floating-point number to the nearest integer.
12. A string is a sequence of characters. Strings are objects of the `String` class.
13. Strings can be *concatenated*, that is, put end to end to yield a new longer string. String concatenation is denoted by the `+` operator.
14. Whenever one of the arguments of the `+` operator is a string, the other argument is converted to a string.
15. If a string contains the digits of a number, you use the `Integer.parseInt` or `Double.parseDouble` method to obtain the number value.
16. Use the `substring` method to extract a part of a string.
17. String positions are counted starting with 0.
18. The `JOptionPane.showInputDialog` method prompts the user for an input string.
19. You must call `System.exit(0)` to exit a program that has a graphical user interface.
20. Wrap `System.in` inside a `BufferedReader` to read input from the console window.
21. When calling the `readLine` method of the `BufferedReader` class, you must tag the calling methods with `throws IOException`.
22. A `char` value denotes a single character. Character constants are enclosed in single quotes.
23. Number variables hold values. Object variables hold references.
24. A copy of an object reference is another reference to the same object.

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.BufferedReader
   readLine
java.io.InputStreamReader
java.lang.Double
   parseDouble
   toString
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
   length
   substring
   toLowerCase
   toUpperCase
java.lang.System
   exit
   in
java.math.BigDecimal
   add
   divide
   multiply
   subtract
```

```
java.math.BigInteger
   add
   divide
   multiply
   subtract
java.text.NumberFormat
   format
   getCurrencyInstance
   getNumberInstance
   setMaximumFractionDigits
   setMinimumFractionDigits
javax.swing.JOptionPane
   showInputDialog
```

Review Exercises

Exercise R3.1. Write the following mathematical expressions in Java.

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

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

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

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

Exercise R3.2. Write the following Java expressions in mathematical notation.

```
dm = m * ((Math.sqrt(1 + v / c) / Math.sqrt(1 - v / c)) - 1);
volume = Math.PI * r * r * h;
volume = 4 * Math.PI * Math.pow(r, 3) / 3;
p = Math.atan2(z, Math.sqrt(x * x + y * y));
```

Exercise R3.3. 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 R3.4. 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 R3.5. Write a test program that executes the following code:

```
Purse myPurse = new Purse();
myPurse.addNickels(3);
```

```
myPurse.addDimes(2);
myPurse.addQuarters(1);
System.out.println(myPurse.getTotal());
```

The program prints the total as `0.6000000000000001`. Explain why. Give a recommendation to improve the program so that users will not be confused.

Exercise R3.6. 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 R3.7. 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 R3.8. Explain the differences between `2`, `2.0`, `'2'`, `"2"`, and `"2.0"`.

Exercise R3.9. Explain what each of the following two program segments computes:

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

and

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

Exercise R3.10. 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 R3.11. True or false? (`x` is an `int` and `s` is a `String`)

- `Integer.parseInt("" + x)` is the same as `x`
- `"" + Integer.parseInt(s)` is the same as `s`
- `s.substring(0, s.length())` is the same as `s`

Exercise R3.12. How do you get the first character of a string? The last character? How do you *remove* the first character? The last character?

Exercise R3.13. 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 R3.14. This chapter contains several recommendations regarding variables and constants that make programs easier to read and maintain. Summarize these recommendations.

Exercise R3.15. What is a `final` variable? Can you define a `final` variable without supplying its value? (Try it out.)

Exercise R3.16. 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";
```

- `x + n * y - (x + n) * y`
- `m / n + m % n`
- `5 * x - n / 5`
- `Math.sqrt(Math.sqrt(n))`
- `(int)Math.round(x)`
- `(int)Math.round(x) + (int)Math.round(y)`
- `s + t`
- `s + n`
- `1 - (1 - (1 - (1 - (1 - n))))`
- `s.substring(1, 3)`
- `s.length() + t.length()`

Exercise R3.17. Explain the similarities and differences between copying numbers and copying object references.

Exercise R3.18. What are the values of `a`, `b`, `c`, and `d` after these statements?

```
double a = 1;
double b = a;
a++;
Purse p = new Purse();
Purse q = p;
p.addNickels(5);
double c = p.getTotal();
double d = q.getTotal();
```

Exercise R3.19. When you copy a `BankAccount` reference, the original and the copy share the same object. That can be significant because you can modify the state of the object through either of the references. Explain why this is not a problem for `String` references.

Programming Exercises

Exercise P3.1. Enhance the `Purse` class by adding methods `addPennies` and `addDollars`.

Exercise P3.2. Add methods `getDollars` and `getCents` to the `Purse` class. The `getDollars` method should return the number of whole dollars in the purse, as an integer. The `getCents` method should return the number of cents, as an integer. For example, if the total value of the coins in the purse is $2.14, `getDollars` returns 2 and `getCents` returns 14.

Exercise P3.3. 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 P3.4. Write a program that prompts the user for two integers and then prints

- The sum
- The difference
- The product
- The average

- The distance (absolute value of the difference)
- The maximum (the larger of the two)
- The minimum (the smaller of the two)

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 (using either a `JOptionPane` or a `BufferedReader`), constructs a `Pair` object, invokes its methods, and prints the results.

Exercise P3.5. 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 P3.6. 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 P3.7. 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, such as

```
final double MILE_TO_KM = 1.609; // from Appendix A7
Converter metersToMiles = new Converter(1000 * MILE_TO_KM);
```

Exercise P3.8. 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 P3.9. 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 P3.10. 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 P3.11. *Giving change.* Implement a program that directs a cashier how to give change. The program has two inputs: the amount due and the amount received from the customer. Compute the difference, and compute the dollars, quarters, dimes, nickels, and pennies that the customer should receive in return.

First transform the difference into an integer balance, denominated in pennies. Then compute the whole dollar amount. Subtract it from the balance. Compute the number of quarters needed. Repeat for dimes and nickels. Display the remaining pennies.

Define a class `Cashier` with methods

- `setAmountDue`
- `receive`
- `returnDollars`
- `returnQuarters`

- `returnDimes`
- `returnNickels`
- `returnPennies`

For example,

```
Cashier harry = new Cashier();
harry.setAmountDue(9.37);
harry.receive(10);
double quarters = harry.returnQuarters(); // returns 2
double dimes = harry.returnDimes(); // returns 1
double nickels = harry.returnNickels(); // returns 0
double pennies = harry.returnPennies(); // returns 3
```

Exercise P3.12. 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 double nextDigit() { . . . }
}
```

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

Exercise P3.13. 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 P3.14. 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 P3.15. *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 P3.16. 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 P3.17. 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 `Year` with methods `getEasterSundayMonth` and `getEasterSundayDay`.