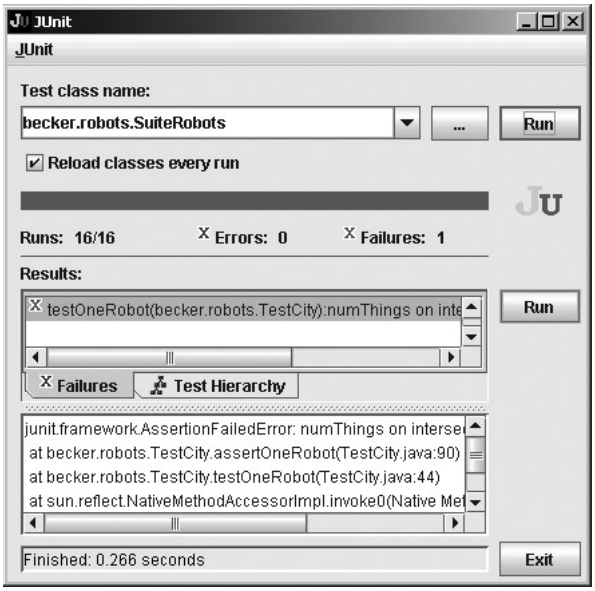
(figure 7-3)

A popular testing tool named `JUnit`



7.2 Using Numeric Types

KEY IDEA

Java's primitive types store values such as `159` *and* `'d'`.

Not everything in Java is an object like a `Robot` or a `Thing`. Integers and the type `int` are the most prominent examples we've seen of a **primitive type**. Primitive types store values such as integers (`159`) and characters (`'d'`), and correspond to how information is represented in the computer's hardware. Primitive types can't be extended and they don't have methods that can be called. In this sense, primitive types distort the design of the language. However, the designers of Java felt it necessary to use primitive types for integers and similar values to increase the execution speed of programs.

Java includes eight primitive types. Six of these store numbers, one stores the Boolean values `true` and `false`, and the last one stores characters.

7.2.1 Integer Types

KEY IDEA

An `int` *can only store values in a certain range.*

Why would Java have six different types to store numbers? Because they differ in the size and precision of the values they store. An `int`, for example, can only store values between `-2,147,483,648` and `2,147,483,647`. This range is large enough to store the net worth of most individuals, but not that of Bill Gates. It's more than enough to store the population of any city on earth, but not the population of the earth as a whole.

To address these issues, Java offers several kinds of integers, each with a different **range**, or number of different values it can store. The ranges of the four integer types are shown in Table 7-1. Variables with a greater range require more memory to store. For programs with many small numbers to store, it makes sense to use a type with a smaller range. Because beginning programmers rarely encounter such programs, we won't need to use `byte` and `short` in this book and will use `long` only rarely.

KEY IDEA

Different types can store different ranges of values.

Type	Smallest Value	Largest Value	Precision
`byte`	-128	127	exact
`short`	-32,768	32,767	exact
`int`	-2,147,483,648	2,147,483,647	exact
`long`	-9,223,372,036,854,775,808	9,223,372,036,854,775,807	exact

(table 7-1)

Integer types and their ranges

7.2.2 Floating-Point Types

Two other primitive types, `float` and `double`, store numbers with decimal places, such as 125.25, 3.14259, or -134.0. They are called **floating-point** types because of the way they are stored in the computer hardware.

Floating-point types can be so large or small that they are sometimes written in **scientific notation**. The number 6.022E23 has two parts, the **mantissa** (6.022) and the **exponent** (23). To convert 6.022E23 to a normal number, write down the mantissa and then add enough zeros to slide the decimal point 23 places to the right. If the exponent is negative, you must add enough zeros to slide the decimal point that many places to the left. For example, 6.022E23 is the same number as 602,200,000,000,000,000,000,000, while 5.89E-4 is the same as 0.000589. Their ranges and precisions are listed in Table 7-2.

KEY IDEA

Scientific notation can be used to express very large or very small numbers.

Type	Smallest Magnitude	Largest Magnitude	Precision
`float`	±1.40239846E-45	±3.40282347E+38	About 7 significant digits
`double`	±4.94065645841246544E-324	±1.79769313486231570E+308	About 16 significant digits

(table 7-2)

The ranges and precisions of the various floating-point types

How big are these numbers? Scientists believe the diameter of the universe is about 1.0E28 centimeters, or 1.0E61 plank units—the smallest unit we can measure. The universe contains approximately 1.0E80 elementary particles such as quarks, the component parts of atoms. So the range of type `double` will certainly be sufficient for most applications.

Floating-point numbers don't behave exactly like real numbers. Consider, for a moment, 1/3 written in decimal: 0.33333.... No matter how many threes you add, 0.33333 won't be exactly equal to 1/3. The situation is similar with 1/10 in binary, the number system computers use. It's impossible to represent 1/10 exactly; the best we can do is to approximate it. The closeness of the approximation is given by the **precision**. `float`s have about 7 digits of precision, while `double`s have about 16 digits. This means, for example, that a `float` can't distinguish between `1.00000001` and `1.00000002`. As far as a `float` is concerned, both numbers are indistinguishable from `1.0`. Another effect is that assigning `0.1` to a `float` and then adding that number to itself 10 times does *not* yield `1.0` but `1.0000001`.

KEY IDEA

Comparing floating-point numbers for equality is usually a bad idea.

The fact that floating-point numbers are only approximations can cause programmers headaches if their programs require a high degree of precision. For beginning programmers, however, this is rarely a concern. One exception, however, is when comparing a `float` or a `double` for equality, the approximate nature of these types may cause an error. For example, the following code fragment appears to print a table of numbers between `0.0` and `10.0`, increasing by `0.1`, along with the squares of those numbers.

```
double d = 0.0;

while (d != 10.0)
{ System.out.println(d + " " + d*d);
  d = d + 0.1;
}
```

The first few lines of the table would be:

```
0.0 0.0
0.1 0.010000000000000002
0.2 0.04000000000000001
0.30000000000000004 0.09000000000000002
0.4 0.16000000000000003
```

Already we can see the problem: `d`, the first number on each line, is not increasing by exactly `0.1` each time as expected. In the fourth line the number printed is only approximately `0.3`.

By the time d gets close to 10.0, the errors have built up. The result is that d skips from 9.99999999999998 to 10.09999999999998 and is never exactly equal to 10.0, as our stopping condition requires. Consequently, the loop keeps printing for a very long time.

The correct way to code this loop is to use an inequality, as in the following code:

```
while (d <= 10.0)
{ ...
}
```

7.2.3 Converting Between Numeric Types

Sometimes we need to convert between different numeric types. In many situations, information is not lost and Java makes the conversion automatically. For example, consider the following statement:

```
double d = 159;
```

Java will implicitly convert the integer 159 to a double value (159.0) and then assign it to d.

The reverse is not true. If assigning a value to another type risks losing information, a **cast** is required. A cast is our assurance to the compiler that we either know from the nature of the problem that information will not be lost, or know that information will be lost and accept or even prefer that result.

KEY IDEA

Casting converts values from one type to another. Sometimes it loses information.

For example, consider the following statements:

```
double d = 3.999;
int i = d;
```

Java will display an error message regarding the second assignment because an integer can't store the decimal part of 3.999, only the 3. If we want to perform this assignment anyway and lose the .999, leaving only 3 in the variable i, we need to write it as follows:

LOOKING AHEAD

Section 7.5.2 discusses a method to round a number rather than truncate it.

```
double d = 3.999;
int i = (int)d;
```

The new part, (int), is the cast. The form of a cast is the destination type placed in parentheses. It can also apply to an entire expression, as in the following statement:

```
int i = (int)(d * d / 2.5);
```

Casting has a high precedence, so you will usually need to use parentheses around expressions.

Assigning from a `double` to an `int` is not the only place information can be lost and a cast required. Information can also be lost assigning values from a `double` to a `float` or from a bigger integer type such as `long` to a smaller type such as `int`.

7.2.4 Formatting Numbers

Java automatically converts a primitive type to a string before concatenating it to another string. This capability allows us to easily print out a mixture of strings and numbers, such as `System.out.println("Age = " + age);` where `age` is an integer.

Automatic conversion to a string does not work as well for `double` values, where we often want to control how many significant digits are printed. For example, the following code might be used in calculating the price of a used car:

```
double carPrice = 12225.00;
double taxRate = 0.15;

System.out.println("Car: " + carPrice);
System.out.println("Tax: " + carPrice * taxRate);
System.out.println("Total: " + carPrice * (1.0 + taxRate));
```

This code gives the following output:

```
Car: 12225.0
Tax: 1833.75
Total: 14058.749999999998
```

These results are far from ideal. We want to see a currency symbol such as $ or £ printed. All of the amounts should have exactly two decimal places, rounding as necessary. The thousands should also be grouped with commas or spaces, depending on local conventions. It's difficult to implement all these details correctly.

Using a `NumberFormat` Object

Fortunately, Java provides a set of classes for formatting numbers, including currencies. These classes all include a method named `format` that takes a number as an argument and returns a string formatted appropriately. Listing 7-4 shows how to use a currency formatting object named `money`. These statements produce formatted output such as the following:

```
Car: $12,225.00
Tax: $1,833.75
Total: $14,058.75
```

Listing 7-4: *Using a currency formatting object*

```
1 double carPrice = 12225.00;
2 double taxRate = 0.15;
3
4 System.out.println("Car: " + money.format(carPrice));
5 System.out.println("Tax: " +
6                    money.format(carPrice * taxRate));
7 System.out.println("Total: " +
8                    money.format(carPrice * (1.0 + taxRate)));
```

FIND THE CODE

ch07/formatNumbers/

A formatting object is not normally obtained by using a constructor. Instead, a **factory method** in the `NumberFormat` class is called. A factory method returns an object reference, as a constructor does. Unlike a constructor, a factory method has the option of returning a subclass of `NumberFormat` that is specialized for a specific task. In this case, the factory method tries to determine the country where the computer is located and returns an object customized for the local currency.

LOOKING AHEAD

Implementing factory methods will be discussed in Chapter 12.

The `NumberFormat` class contains the `getCurrencyInstance`, `getNumberInstance`, and `getPercentInstance` factory methods, along with several others. The `getCurrencyInstance` factory method can be used by importing `java.text.NumberFormat` and including the following statement before line 4 in Listing 7-4.

KEY IDEA

Factory methods help you obtain an object already set up for a specific situation.

```
NumberFormat money = NumberFormat.getCurrencyInstance();
```

A formatter for general numbers can be obtained with the `getNumberInstance` factory method. It can be customized to format numbers with a certain number of decimal places and to print grouping characters. Consider the following example:

```
NumberFormat f = NumberFormat.getNumberInstance();
f.setMaximumFractionDigits(4);
f.setGroupingUsed(true);
System.out.println(f.format(3141.59265359));
```

These statements will print the value `3,141.5927`—the value rounded to four decimal places with an appropriate character (in this case, a comma) used to group the digits.

Columnar Output

Programs often produce lots of numbers that are most naturally formatted in columns. Even with the program to calculate the tax for a car purchase, aligning the labels and numbers vertically makes the information easier to read.

One of the easiest approaches uses the `printf` method in the `System.out` object. It was added in Java 1.5, and is not available in earlier versions of Java.

KEY IDEA

`printf`'s format string says how to format the other arguments.

The `printf` method is unusual in that it takes a variable number of arguments. It always takes at least one, called the **format string**, that includes embedded codes describing how the other arguments should be printed.

Here's an example where `printf` has three arguments.

```
System.out.printf("%-10s%10s", "Car:", money.format(carPrice));
```

FIND THE CODE

ch07/formatNumbers/

The first argument is the format string. It includes two **format specifiers**, each one beginning with a percent (`%`) sign and ending with a character indicating what kind of data to print. The first format specifier is for the second argument; the second specifier is for the third argument. Additional specifiers and arguments could easily be added.

In each case, the `s` indicates that the argument to print should be a string. The `10` instructs `printf` to print the string in a field that is 10 characters wide. The minus sign (`-`) in one says to print that string **left justified** (starting on the left side of the column). The specifier without the minus sign will print the string **right justified** (on the right side of the column).

This line, as specified, does not print a newline character at the end; thus, any subsequent output would be on the same line. We could call `println()` to end the line, or we could add another format specifier. The specifier `%n` is often added to the format string to begin a new line. It does *not* correspond to one of the arguments.

Table 7-3 gives several examples of the most common format specifiers and the results they produce. A `d` is used to print a decimal number, such as an `int`. An `f` is used to print a floating-point number, such as a `double`. In addition to the total field width, it specifies how many decimal places to print. More examples and a complete description are available in the online documentation for the `java.util.Formatter` class.

(table 7-3)

Examples of common format specifiers; dots signify spaces

Format Specifier and Argument	Result
`"%-10s", "Car:"`	`Car:`⋅⋅⋅⋅⋅⋅
`"%10s", "Car:"`	⋅⋅⋅⋅⋅⋅`Car:`
`"%10d", 314`	⋅⋅⋅⋅⋅⋅⋅`314`
`"%10.4f", 3.1415926`	`3.1416`⋅⋅⋅⋅
`"%-10.4f", 3.1415926`	⋅⋅⋅⋅`3.1416`

The `printf` method has many other options that are documented in the `Formatter` class. Discussing them further, however, is beyond the scope of this book.

7.2.5 Taking Advantage of Shortcuts

KEY IDEA

`i++` *is a shortcut for* `i = i + 1`.

Java includes a number of shortcuts for some of the most common operations performed with numeric types. For example, one of the most common is to add 1 to a variable. Rather than writing `ave = ave + 1`, Java permits the shortcut of writing `ave++`. A similar shortcut is writing `ave--` in place of `ave = ave - 1`.

It is also common to add the result of an expression to a variable. For example, the following is `SimpleBot`'s `move` method as written in Listing 6-6:

```
public void move()
{ this.street = this.street + this.strOffset();
  this.avenue = this.avenue + this.aveOffset();
  Utilities.sleep(400);
}
```

Instead of repeating the variable on the right side of the equal sign, we can use the `+=` operator, which means to add the right side to the value of the variable on the left, and then store the result in the variable on the left. More precisely, *«var»* `+=` *«expression»* means *«var»* `=` *«var»* `+ (`*«expression»*`)`. The parentheses are important in determining what happens if *«expression»* contains more than a single value. The following example is equivalent to the previous code:

```
public void move()
{ this.street += this.strOffset();
  this.avenue += this.aveOffset();
  Utilities.sleep(400);
}
```

There are also `-=`, `*=`, and `/=` operators. They are used much less frequently but behave the same as `+=` except for the change in numeric operation.

7.3 Using Non-Numeric Types

Variables can also store information that is not numeric, using the types `boolean`, `char`, and `String`.

7.3.1 The `boolean` Type

The `boolean` type is used for `true` and `false` values. We have already seen Boolean expressions used to control `if` and `while` statements, and as a temporary variable and the return type in predicates (see, for example, Listing 5-3). We have also explored using `boolean` values in expressions (see Section 5.4).

Instance variables, named constants, and parameter variables can also be of type `boolean`. For example, a Boolean instance variable can store information about whether a robot is broken. The robot might consult that variable each time it is asked to move, and only move if it has not been previously broken.

LOOKING BACK

A Boolean temporary variable is used in `rightIsBlocked`, *section 5.2.5.*

```
public class SimpleBot extends Paintable
{ private int avenue;
  private int street;
  private boolean isBroken = false;
  ...

  public void breakRobot()
  { this.isBroken = true;
  }

  public void move()
  { if (!this.isBroken)
    { this.avenue = ...
      this.street = ...
    }
  }
  ...
}
```

7.3.2 The Character Type

A single character such as `a`, `Z`, `?`, or `5` can be stored in a variable of type `char`. These include the characters you type at the keyboard—and many more that you can't type directly. Like the other primitive types, the `char` type may be used for instance variables, temporary variables, parameter variables, and named constants, and may be returned from queries.

One use for characters is to control a robot from the keyboard. `Sim`, a superclass of `Robot`, has a protected method named `keyTyped` that is called each time a key is typed, yet it does nothing. The method has a `char` parameter containing the character that was typed. By overriding the method, we can tell a robot to move when '`m`' is typed, turn right when '`r`' is typed, and so on. The `KeyBot` class in Listing 7-5 defines such a robot. The same technique can be used in subclasses of `Intersection` and `Thing` because they all descend from `Sim`—the class implementing `keyTyped`. (When running a program using this feature, you must click on the image of the city before it will accept keystrokes. The image will have a black outline when it is ready to accept keystrokes.)

Listing 7-5: *A robot that responds to keystrokes*

```
1  import becker.robots.*;
2
3  public class KeyBot extends RobotSE
4  {
5    public KeyBot(City c, int str, int ave, Direction dir)
6    { super(c, str, ave, dir);
7    }
8
9    protected void keyTyped(char key)
10   { if (key == 'm' || key == 'M')
11     { this.move();
12     } else if (key == 'r' || key == 'R')
13     { this.turnRight();
14     } else if (key == 'l' || key == 'L')
15     { this.turnLeft();     // Watch out. The above test uses
16                            // a lowercase 'L', not a "one".
17   }
18 }
```

FIND THE CODE

ch07/keyBot/

The parameter, `key`, is compared to the letters '`m`', '`r`', and '`l`' in lines 10, 12, and 14. In each case, if the comparison is true (that is, the parameter contains an '`m`', '`r`', or '`l`'), an action is taken. If a different key is pressed, the robot does nothing. A slightly enhanced version of this method is implemented in the `RobotRC` class. You can extend `RobotRC` anytime you want to use the keyboard as a remote control (RC) for a robot.

KEY IDEA

Override `keyTyped` *to make a robot that can be controlled from the keyboard.*

The `'m'`, `'r'`, and `'l'` are character literals. To write a specific character value, place the character between two single quotes. What if you want to compare a value to a single quote? Placing it between two other single quotes (`''''`) confuses the compiler, causing an error message. The solution is to use an **escape sequence**. An escape sequence is an alternative way to write characters that are used in the code for other purposes. The escape sequence for a single quote is `\'` (a backslash followed by a single quote). All escape sequences begin with a backslash. The escape sequence is placed in single quotes, just like any other character literal. Table 7-4 shows some common escape sequences, many of which have their origins in controlling printers.

KEY IDEA

Some characters have special meaning to Java. They have to be written with an escape sequence.

The last escape sequence, `\u`*dddd*, is used for representing characters from a wide range of languages, and includes everything from accented characters to Bengali characters to Chinese ideograms. You can find more information online at *www.unicode.org*. Unfortunately, actually using these characters requires corresponding fonts on your computer.

(table 7-4)

Character escape sequences

Sequence	Meaning
`\'`	Single quote
`\"`	Double quote
`\\`	Backslash
`\n`	Newline—used to start a new line of text when printing at the console
`\t`	Tab—inserts space so that the next character is placed at the next **tab stop**. Each tab stop is a predefined distance from the previous tab stop.
`\b`	Backspace—moves the cursor backwards over the previously printed character
`\r`	Return—moves the cursor to the beginning of the current line
`\f`	Form feed—moves the cursor to the top of the next page in a printer
`\u`*dddd*	A Unicode character, each *d* being a hexadecimal digit (0–9, a–f, A–F)

7.3.3 Using Strings

Strings of characters such as "Hello, karel!" are used frequently in Java programs. Strings are stored, appropriately, in variables of type `String`. A string can hold thousands of characters or no characters at all (the empty string). These characters can be the familiar ones found on the keyboard or those specified with escape characters, as shown in Table 7-4.

`String` is *not* a primitive type. In fact, it is a class just as `Robot` is a class. On the other hand, strings are used so often that Java's designers included special support for them that other classes do not have—so much special support that it sometimes feels like strings are primitive types.

Special Java Support for Strings

KEY IDEA

Java provides special support for the `String` *class.*

The special support the `String` class enjoys from the Java compiler falls into three categories:

- Java will automatically construct a `String` object for each sequence of characters between double quotes; that is, Java has literal values for strings just like it has literal values for integers (`5`, `-259`), doubles (`3.14159`), and Booleans (`true`).
- Java will "add" two strings together with the plus operator to create a new string consisting of one string followed by the other. This is called **concatenation**.
- Java will automatically convert primitive values and objects to strings before concatenating them with a string.

Listing 7-6 shows several examples of this special support. The program uses `System.out.println` to print the strings, as we did in Section 6.6.1. The difference here is the manipulations of the strings before they are printed.

FIND THE CODE

ch07/stringDemo/

Listing 7-6: *A simple program demonstrating built-in Java support for the* `String` *class*

```
1  import becker.robots.*;
2
3  public class Main
4  {
5    public static void main(String[] args)
6    { String greeting = "Hello";
7      String name = "karel";
8
9
10     System.out.println(greeting + "," + name + "!");
11
12
13
14     System.out.println("Did you know that 2*PI = " + 2*Math.PI + "?");
15
16     City c = new City();
17     Robot karel = new Robot(c, 1, 2, Direction.SOUTH);
18     System.out.println("c=" + c);
19   }
20 }
```

A `String` object is created automatically from the string literal "Hello"

Four strings are concatenated using the "+" operator to produce a single string, "Hello, karel!"

The primitive value resulting from this expression is automatically converted to a string and concatenated using the plus operator

The object referenced by `c` is automatically converted to a string by calling its `toString` method

Program output:

```
Hello, karel!
Did you know that 2*PI = 6.283185307179586?
c=becker.robots.City[SimBag[robots=[becker.robots.Robot
[street=1, avenue=2, direction=SOUTH, isBroken=false, numThings
InBackpack=0]], things=[]]]
```

In lines 6 and 7, two `String` objects are created using the special support the Java language provides for strings. These lines would look more familiar if they used a normal constructor, which works as expected:

```
String greeting = new String("Hello");
String name = new String("karel");
```

Line 14 contains an expression that is evaluated before it is passed as an argument to `println`. The normal rules of evaluation are used: multiplication has a higher precedence than addition, so `2*Math.PI` is evaluated first. Then, two string additions, or concatenations, are performed left to right. Because the left and right sides of the first

addition operator do not have the same type, the less general one (the result of `2*Math.PI`) is converted to a string before being "added" to the other operand.

Finally, when Java converts an object to a string, as it does in line 18, it calls the method named `toString`, which every class inherits from `Object`.

Overriding `toString`

KEY IDEA

Every class should override `toString` *to provide meaningful information.*

Java depends on the fact that every object has a `toString` method that can be called to provide a representation of the object as a string. The default implementation, inherited from the `Object` class, only prints the name of the class and a number identifying the particular object. To be useful, the method should be overridden in classes you write. The information it presents is often oriented to debugging, but it doesn't have to be.

The standard format for such information is the name of the object's class followed by an open bracket, "`[`". Information relevant to the object follows, and then ends with a closing bracket, "`]`". This format allows objects to be nested. For example, when the `City` object is printed, we see that it prints the `Robot` and `Thing` objects it references. Each of these, in turn, print relevant information about themselves, such as their location.

Listing 7-7 shows a `toString` method that could be added to the `SimpleBot` class shown in Listing 6-6.

toString

Listing 7-7: *A sample* `toString` *method*

```
1  public class SimpleBot extends Paintable
2  { private int street;
3    private int avenue;
4    private int direction;
5
6    // Constructor and methods are omitted.
7
8   /** Represent a SimpleBot as a string. */
9   public String toString()
10  { return "SimpleBot" +
11        " [street=" + this.street +
12        ", avenue=" + this.avenue +
13        ", direction=" + this.direction +
14        "]";
15  }
16 }
```

Querying a String

The `String` class provides many methods to query a `String` object. These include finding out how long a string is, whether two strings start the same way, the first location of a particular character, and so on. The most important of these queries are shown in Table 7-5.

(table 7-5)

Methods that query a string

Method	Description
`char charAt(int index)`	Returns the character at the location specified by the **index**. The index is the position of the character—an integer between 0 (the first character) and one less than the length of the string (the last character).
`int compareTo(String aString)`	Compares this string to `aString`, returning a negative integer if this string is lexicographically smaller than `aString`, 0 if the two strings are equal, and a positive integer if this string is lexicographically greater than `aString`.
`boolean equals(Object anObject)`	Compares this string to another object (usually a string). Returns `true` if `anObject` is a string containing exactly the same characters in the same order as this string.
`int indexOf(char ch)`	Returns the index within this string of the first occurrence of the specified character. If the character is not contained within the string, `-1` is returned.
`int indexOf(char ch, int fromIndex)`	Returns the index within this string of the first occurrence of the specified character, starting the search at `fromIndex`. If no such character exists, `-1` is returned.
`int indexOf(String substring)`	Returns the index of the first character of the first occurrence of the given substring within this string. If the given substring is not contained within this string, `-1` is returned.
`int lastIndexOf(char ch)`	Returns the index of the last occurrence of the given character within this string. If the given character is not contained within this string, `-1` is returned.
`int length()`	Returns the number of characters contained in this string.
`boolean startsWith(String prefix)`	Returns `true` if this string starts with the specified prefix.

KEY IDEA

Characters in strings are numbered starting at position `0`.

The `charAt` and `indexOf` methods in Table 7-5 refer to a character's index, or position within the string. In the string "Hello", 'H' is at index 0, 'e' is at index 1, and 'o' is at index 4. For example, if the variable `greeting` refers to "Hello", then `greeting.charAt(1)` returns the character 'e'.

It may seem strange for strings to begin indexing at zero, but this is common in computer science. We have already seen it in the robot cities, where streets and avenues begin with zero. We'll see it again in upcoming chapters, where collections of values are indexed beginning with zero.

KEY IDEA

>, >=, <, and <= don't work for strings.

When `a` and `b` are primitive types, we can compare them with operators such as `a == b`, `a < b`, and `a >= b`. For reference types such as `String`, only the == and `!=` operators work—and they do something different than you might expect.

KEY IDEA

Use the `equals` *method to compare strings for equality.*

Instead of ==, compare two strings for equality with the `equals` method. It returns true if every position in both strings has exactly the same character.

```
if (oneString.equals(anotherString))
{ System.out.println("The strings are equal.");
}
```

Instead of `!=`, use a Boolean expression, as follows:

```
!oneString.equals(anotherString)
```

KEY IDEA

Use `compareTo` *to compare strings for order.*

The string equivalent to less than and greater than is the `compareTo` method. It can be used as shown in the following code fragment:

```
String a = ...
String b = ...
if (a.compareTo(b) < 0)
{  // a comes before b in the dictionary
} else if (a.compareTo(b) > 0)
{  // a comes after b in the dictionary
} else  // if (a.compareTo(b) == 0)
{  // a and b are equal
}
```

The `compareTo` method determines the **lexicographic order** of two strings—essentially, the order they would have in the dictionary. To determine which of two strings comes first, compare the characters in each string, character by character, from left to right. Stop when you reach the end of one string or a pair of characters that differ. If you stop because one string is shorter than the other, as is the case with "`hope`" and "`hopeful`" in Figure 7-4, the shorter string precedes the longer string. If you stop because characters do not match, as is the case with "`f`" and "`l`" in "`hopeful`" and "`hopeless`", then compare the mismatched characters. In this case "`f`" comes before "`l`", and so "`hopeful`" precedes "`hopeless`" in lexicographic order.

```
hope
hopeful
hopeless
```

(figure 7-4)

Lexicographic ordering

If the strings have non-alphabetic characters, you may consult Appendix D to determine their ordering. For example, a fragment of the ordering is as follows:

```
! " # … 0 1 2 … 9 : ; < … A B C … Z [ \ … a b c … z { | }
```

This implies that "`hope!`" comes before "`hopeful`" because `!` appears before `f` in the previous ordering. Similarly, "`Hope`" comes before "`hope`".

Transforming Strings

Other methods in the `String` class do not answer questions about a given string, but rather return a copy of the string that has been transformed in some way. For example, the following code fragment prints "`Warning WARNING`"; `message2` is a copy of `message1` that has been transformed by replacing all of the lowercase characters with uppercase characters.

```
String message1 = "Warning";
String message2 = message1.toUpperCase();
System.out.println(message1 + " " + message2);
```

The designers of the `String` class had two options for the `toUppercase` method. They could have provided a command that changes all of the characters in the given string to their uppercase equivalents. The alternative is a method that makes a copy of the string, changing each lowercase letter in the original string to an uppercase letter in the copy.

KEY IDEA

An immutable class is one that does not provide methods to change its instance variables.

The designers of the `String` class consistently chose the second option. This makes the `String` class immutable. After a string is created, it cannot be changed. The methods given in Table 7-6, however, make it easy to create copies of a string with specific transformations. The `StringBuffer` class is similar to `String`, but includes methods that allow you to modify the string instead of creating a new one.

The `substring` method is slightly different. Its transformation is to extract a piece of the string, returning it as a new string. For example, if `name` refers to the string "`Karel`", then `name.substring(1,4)` returns "`are`". Recall that strings are indexed beginning with 0, so the character at index 1 is `a`. The second index to `substring`, 4 in this example, is the index of the first character *not* included in the substring.

(table 7-6)

Methods that return a transformed copy of a string

Method	Description
`String replace(char oldChar, char newChar)`	Returns a copy of this string that has all occurrences of `oldChar` replaced with `newChar`.
`String substring(int beginIndex, int endIndex)`	Returns a new string containing all the characters between `beginIndex` and `endIndex-1`, inclusive. The character at `endIndex` is the first character not included in the new string.
`String toLowerCase()`	Returns a copy of this string that has all the uppercase characters replaced with their lowercase equivalents.
`String toUpperCase()`	Returns a copy of this string that has all the lowercase characters replaced with their uppercase equivalents.
`String trim()`	Returns a copy of this string that has all white space (such as space, tab, and newline characters) removed from the beginning and end of the string.

Example: Counting Vowels

As an illustration of what you can do with strings, let's write a program that counts the number of vowels (a, e, i, o, u) in a string. As a test, we'll use the famous quotation from Hamlet, "To be, or not to be: that is the question." The expected answer is 13.

To begin solving this problem, let's think about how to solve it without a computer. One straightforward method is to look at each letter, proceeding from left to right. If the letter is a vowel, we can put a tick mark on a piece of paper. When we get to the end, the number of ticks corresponds to the number of vowels. Our program can adopt a similar strategy by using a variable to record the number of vowels.

This illustration will require us to examine individual letters in a string and compare letters to other letters. We don't have experience solving these kinds of problems, so let's proceed by solving a series of simpler problems. First, let's print the individual letters in the quotation. This shows that we can process the letters one at a time. After mastering that, let's count the number of times a single vowel, such as 'o', occurs. Finally, after solving these subproblems, we'll count all the vowels.

To print all the letters in the quotation, we must access each individual letter. According to Table 7-5, the `charAt` method will return an individual character from a string. However, it needs an index, a number between 0 and one less than the length of the string. Evidently, the `length` method will also be useful. To obtain the numbers between 0 and the length, we could use a `for` loop. We'll start a variable named `index` at 0 and increment it by 1 each time the loop is executed until `index` is just less than

the length. These ideas are included in the following Java program that prints each character in the quotation, one character per line.

```
 1  public static void main(String[] args)
 2  { String quotation = "To be, or not to be: that is the question.";
 3
 4    // Loop over each letter in the quotation.
 5    for (int index = 0; index < quotation.length(); index++)
 6    { // Examine one letter in the quotation.
 7      char ch = quotation.charAt(index);
 8      System.out.println(ch);
 9    }
10  }
```

Notice that the `for` loop starts the index at `0`, the first position in the string. The loop continues executing as long as the index is *less* than the length of the string. As soon as it equals the length of the string, it's time to stop. For example, Figure 7-5 illustrates a string of length 5, but its largest index is only 4. Therefore, the appropriate test to include in the `for` loop is `index < quotation.length()`.

KEY IDEA

A string of length 5 *has indices numbered* 0 *to* 4.

Index:	0	1	2	3	4
Characters:	T	o		b	e

(figure 7-5)

A string with its index positions marked

To modify this program to count the number of times 'o' appears, we can replace the `println` with an `if` statement and add a counter. The call to `println` in line 14 concatenates the value of our `counter` variable with two strings to make a complete sentence reporting the results. The modifications are shown in bold in the following code:

```
 1  public static void main(String[] args)
 2  { String quotation = "To be, or not to be: that is the question.";
 3
 4    int counter = 0;          // Count number of os.
 5    // Loop over each letter in the quotation.
 6    for (int index = 0; index < quotation.length(); index++)
 7    { // Examine one letter in the quotation.
 8      char ch = quotation.charAt(index);
 9      if (ch == 'o')
10      { counter += 1;
11      }
12    }
13
14    System.out.println("There are " + counter + " occurrences of 'o'.");
15  }
```

The last step is to count *all* the vowels instead of only the os. A straightforward approach is to add four more `if` statements, all similar to the one in lines 9–11. However, when we consider that other quotations might include uppercase vowels (totaling 10 `if` statements), looking for an alternative becomes attractive.

We can reduce the number of tests if we first transform the quote using `toLowerCase`, as shown in line 3 of Listing 7-8. This assures us that all vowels will be lowercase.

KEY IDEA

`indexOf` searches a string for a particular character.

The `indexOf` method shown in Table 7-5 offers an interesting possibility. It will search a string and return the index of the first occurrence of a given character. If the character isn't there, `indexOf` returns -1. Suppose we take a letter from our quotation and search for it in a string that has only vowels. If the letter from the quotation is a vowel, it will be found and `indexOf` will return a 0 or larger. If it's not there, `indexOf` will return -1. This idea is implemented in Listing 7-8. The changes from the previous version are again shown in bold.

FIND THE CODE

ch07/countVowels/

Listing 7-8: *Searching a string to count the number of vowels*

```
1  public static void main(String[] args)
2  { String quotation = "To be, or not to be: that is the question";
3    String lowerQuote = quotation.toLowerCase();
4    String vowels = "aeiou";
5
6    int counter = 0;        // Count the number of vowels.
7    // Loop over each letter in the quotation.
8    for (int index = 0; index < lowerQuote.length(); index++)
9    { // Examine one letter in the quotation.
10     char ch = lowerQuote.charAt(index);
11     if (vowels.indexOf(ch) >= 0)
12     { counter += 1;
13     }
14   }
15
16   System.out.println("There are" + counter + "vowels.");
17 }
```

7.3.4 Understanding Enumerations

KEY IDEA

Java version 1.5 or higher is required to use this feature.

Programmers often need a variable that holds a limited set of values. For example, we may you need to store a person's gender—either male or female. For this we need only two values.