6 Basic Data Types and Expressions

Overview

In this chapter you will learn about decimal number values, character values, long values, and more String methods, to help develop software for managing a car repair shop. You will also have a light introduction to graphical components. The first five sections complete the coverage of all the language features you need for the first half or so of each of Chapters Eight through Twelve. In particular, if you want to use disk files for input and output, you can read and understand the first three sections of Chapter Nine and of Chapter Twelve on disk files after you complete Section 6.5.

- Sections 6.1-6.2 discuss the double type of value (numbers with decimal points).
- Sections 6.3-6.4 introduce more String methods using chars (character values).
- Sections 6.5-6.6 give some details on conversions and casts between types of values, as well as an introduction to the Math class and JTextArea objects.
- Sections 6.7-6.8 complete the development of the RepairShop software, with the main application class depending on six other classes: Queue, IO, RepairOrder, View, String, and System (only the last two are from the Sun standard library).

6.1 Double Values, Variables, And Expressions

Suppose you have a job developing a program to help a car repair shop schedule each day's repair jobs. The shop accepts appointments and walk-ins on a first-come-first-served basis. The shop foreman enters each repair job into the computer as it is booked, putting it at the end of a list of such jobs. When a time slot becomes available to work on a car, the next job at the front of the list is removed from the list and worked on. Walk-ins during the day are added to the end of the list. This kind of list is called a **queue**.

Your RepairShop program is to report on the total number of jobs waiting and the total estimated time to complete those jobs. An updated report is to be made after each change in the list. So this software deals with several different kinds of objects, including:

1. A virtual repair job, representing a single work order.
2. A virtual input device, representing the keyboard.
3. A virtual output device, representing the screen.
4. A virtual queue, storing data for several repair jobs on a first-in-first-out basis.

You will need to store the number of hours for a single work order as a decimal number, e.g., 1.25 hours for 1 hour 15 minutes. And you will need to read in a line of input containing several words and separate out each word from the rest. So you need more information on working with decimal numbers and Strings. This will be supplied in the next few sections, then we will come back to the RepairShop software and complete it.

Decimal Numbers

You may declare a variable as type **double**. It can then hold decimal numbers such as 53.172 and -0.00005. Java sets aside a 64-bit storage space for its value, stored in scientific notation (<u>double</u> the amount for ints, hence the name). That is enough space for 15-decimal-place precision, with some space left over to store an exponent of 10 up to about the 307th power. In short, you can store a 306-digit number with 15-digit precision. The five numeric operators for doubles are `+` for plus, `*` for times, `-` for minus, `/` for divided-by, and `%` for the remainder. For example, `13.0 / 4.0` is 3.25, the result of "long division". `8.0 % 2.5` is 0.5 and `(-8.0) % 2.5` is -0.5, similar to int values.

The six comparison operators can be used for both int values and double values: The expression `x < y` means that x is less than y. Similarly, `x > y` means that x is greater than y, `x <= y` means that x is less than or equal to y, and `x >= y` means that x is greater than or equal to y. `x != y` means that x is not equal to y and `x == y` means that x is equal to y (the double equals sign is needed to distinguish it from the assignment operator).

If you want to obtain a double value from the user, you can convert the string of characters `s` that `showInputDialog` returns to a double value with the following statement. It uses the `parseDouble` class method from the Double class in the `java.lang` package, which is analogous to `Integer.parseInt`. This class method accepts either normal decimal form (e.g., -4176000.0 or 0.000000275) or **scientific notation** (their equivalents -4.176E+6 or 2.75E-7). Note that the 'E' in this form stands for "times ten to the power": 6.3E+4 is 6.3 times ten to the power 4, i.e., 63000.

```
   double x = Double.parseDouble (s);
```

The program can crash if the user's input contains letters or is otherwise ill-formed. However, the `Double.parseDouble` method (which is new with Java version 1.2) will tolerate blanks before the numeral, whereas `Integer.parseInt` will not.

Promotions and casts

If you combine an int value with a double value using an operator, the runtime system automatically **promotes** the whole number to the corresponding decimal form. The same thing happens if you assign an int value to a double variable. So if you have two int variables `x` and `y`, then `(1.0 * x) / y` gives a precise answer for the quotient.

You cannot assign a double value to an int variable without saying that you are doing it. You do this with a **cast**, which is the type name in parentheses. For instance, if `dub` is a double variable and `ent` is an int variable, then `dub = ent` is legal, but `ent = dub` is not legal. However, `ent = (int) dub` is legal. This assignment discards the part after the decimal point in `dub`'s value (3.8 is changed to 3, and -3.8 is changed to -3). In general, you are allowed to make a cast from any numeric value to any other kind of numeric value, but you may lose some of the value.

Sentinel logic

A standard pattern for user interaction is to ask for a particular kind of value over and over again, using that value in a new calculation. When the user wants to stop the program, the user enters a special signal value such as 0 or -1 that cannot occur as a value to use in the calculation. Such a signal is called a **sentinel value**.

The GrowthRates class in Listing 6.1 (see next page) illustrates this **sentinel-controlled loop pattern**. It asks the user for an interest rate (such as 6% or 15%, but without the percent sign). Then it calculates how many years it would take for money to grow at that interest rate to be twice as much and reports the answer. The sentinel values are all the nonpositive numbers (i.e., an "interest rate" that is not positive terminates the program).

Listing 6.1 illustrates the **class pattern for reusable software** (as opposed to programs designed to exercise your understanding of language features): The main method does little more than create an object that does the real job. On the assumption that the GrowthRates software is valuable, you do not want to limit the use of the software to being run from the command line. To promote reusability, you write it so that any software can create a GrowthRates object and have it compute doubling time. The main method is there only for testing purposes. This listing compiles as one file -- It is legal to declare a class without "public" if you do not intend to call its methods from other classes. The file name must be GrowthRates.java since GrowthRates is the public class in the file.

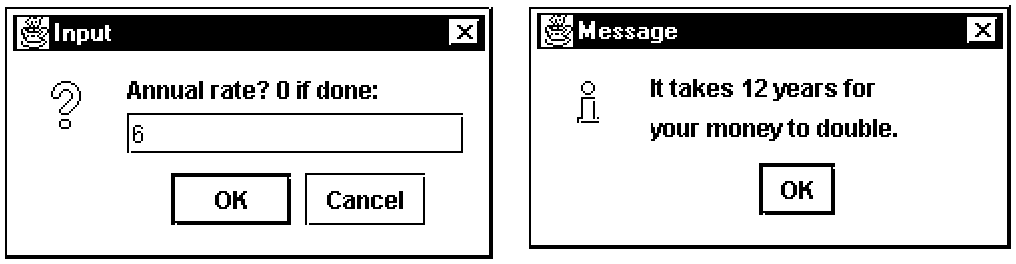
Listing 6.1 The GrowthRates object class

```
import javax.swing.JOptionPane;

class GrowthRatesTester
{
   public static void main (String[ ] args)
   {  JOptionPane.showMessageDialog (null,
                "Calculating growth for various interest rates");
      new GrowthRates();
      System.exit (0);
   }  //======================
}  //#######################################################

public class GrowthRates extends Object
{
   /** Calculate time to double your money at a given rate. */

   public GrowthRates()
   {  String input = JOptionPane.showInputDialog
                ("Annual rate? 0 if done:");
      double rate = Double.parseDouble (input);
      while (rate > 0.0)
      {  JOptionPane.showMessageDialog (null,
                 "It takes " + yearsToDouble (rate)
                 + " years for \nyour money to double.");
         input = JOptionPane.showInputDialog
                ("Another rate (0 when done):");
         rate = Double.parseDouble (input);
      }
   }  //======================

   /** Precondition:  interestRate is positive. */

   public int yearsToDouble (double interestRate)
   {  double balance = 1.0;
      int count = 0;
      while (balance < 2.0)
      {  balance = balance * (1.0 + interestRate / 100.0);
         count++;
      }
      return count;
   }  //======================
}
```

The calculation requires repeatedly multiplying one year's balance by 1.06 if the `rate` is 6%, by 1.15 if the `rate` is 15%, etc. The command `count++` means that `count` is increased by 1. This logic illustrates the common **count-cases looping action**: If you initialize a counter variable to zero before the loop begins, and you increment it each time through the loop, then its value when you exit the loop will be the number of times that the body of the loop was executed.

The following is the sequence of phrases you will see on the screen when you run this program and enter the boldfaced values:

```
   Calculating growth for various interest rates
   Annual rate? 0 if done: 6
   It takes 12 years for your money to double.
   Another rate (0 when done): 9.1
   It takes 8 years for your money to double.
   Another rate (0 when done): 12.3
   It takes 6 years for your money to double.
   Another rate (0 when done): -2
```



Annual rate? 0 if done:

6

It takes 12 years for your money to double.

Figure 6.1 SCREEN SHOTS of dialog boxes for GrowthRates

Programming Style When you have a program that interacts with the user by keyboard and screen, it is good style to begin with a line or two on the screen indicating the purpose of the program. That way, the user knows right away if he/she is running the desired program. The first statement of Listing 6.1 illustrates this principle.

Special assignment operators

The assignment to the `balance` variable in Listing 6.1 can be written as follows:

```
   balance  *=  1.0 + interestRate / 100.0;
```

This is an example of a **special assignment operator**. You can write **`x *= y`** instead of `x = x * y`. The former executes somewhat faster and seems a little easier to understand once you get used to it. Similarly, **`x += y`** means `x = x + y`, **`x -= y`** means `x = x - y`, **`x /= y`** means `x = x / y`, and **`x %= y`** means `x = x % y`. For instance, the following two statements have exactly the same effect:

```
   total = total + someGuy.getBirthYear();
   total += someGuy.getBirthYear();
```

Language elements
Five additional assignment operators are +=, -=, *=, /=, and %=.
A numeric type of variable in Java is double. These values are not objects; they are normally written as two sequences of digits separated by a decimal point, with an optional minus sign. You may use the arithmetic, assignment, and comparison operators that ints have.
If you want to assign a double value to an int variable, you must put (int) in front of the double expression. This truncates the value after the decimal point.
You may replace "public class" by just "class" if you do not use it except for testing.

Exercise 6.1 Write an application program to ask the user for three decimal numbers and then print their average. Assume numeric input is well-formed.
Exercise 6.2 Write an application program to ask the user for the exchange rate for German marks and the German price per liter (0.264 gallons) of gasoline, then say what the price is in dollars per gallon. Assume numeric input is well-formed.
Exercise 6.3 Find three phrases in the Interlude before Chapter Four that can reasonably be rewritten to use a special assignment operator.
Exercise 6.4 (harder) Write an application program to ask the user for a number of days and print the number of seconds in that many days. Also print the number of minutes and the number of hours. Do not put any numeric value greater than 60 in your program.
Exercise 6.5* Revise GrowthRates to compound monthly. For instance, if the rate is 6%, multiply by 1.005 each month, then tell the number of years and months to double.

Exercise 6.6* Write an application program to ask the user for a (double) temperature in Kelvin degrees and convert it to Fahrenheit. Also print the Fahrenheit temperature at which water is inert. Note: It helps to know that water boils at 373.15 degrees Kelvin, freezes at 273.15 degrees Kelvin, and is inert at absolute zero Kelvin.
Exercise 6.7* Write an application program to ask the user for the ages of three people and then print the age of the youngest.
Exercise 6.8** Write an application program to ask the user for the amount paid for a purchase and also for the amount of money offered in payment. Print out the number of pennies, nickels, dimes, and quarters that the user receives in change. The change should produce the least possible number of coins. But print "can't do that" if the change is not in the range 0 through 99 cents. Hint: `quarters = (offered - paid) / 25`.

6.2 Creating Your Own Library Classes

The JOptionPane method calls and the string-to-number conversions obscure the basic logic of a program. It is difficult enough for a programmer to work out complex logic, as in Listing 6.1 with its two while-statements, and difficult enough for other readers to make sense of the logic, without the intrusion of long phrases that perform a simple task.

Programming Style A broadly-accepted rule of good style in Java is, if you find that several lines of logic appear in several different places in your programs, and if those lines act together to accomplish a single well-defined task, then write a method to contain those lines and use calls of that method instead. Put the method in your personal library of classes if you expect it to be needed in several programs. It is good to treat even one line of logic this way if it is used very frequently and is rather lengthy or arcane.

The repair shop client wants the name of the shop displayed on each message dialog box instead of just "Message". Also, the client does not like the information icon; you are to get rid of it. A variant of the `showMessageDialog` method lets you do these things, but it makes the method call even wordier: You add two parameters, one the title and one having the value `JOptionPane.PLAIN_MESSAGE` (which omits the icon).

If you were to make this change throughout the RepairShop software, and then sell this software to another repair shop, you would have to make many changes in your software. Clearly you need to define methods that encapsulate keyboard input and messages.

The IO class

The solution is to develop a class that adapts the general-purpose JOptionPane methods especially for use with this RepairShop software. You put the `showMessageDialog` call in a method in this library class of your own, which you name perhaps IO. Now you can just change a few words in one place to use it for another repair shop. You also put three kinds of `showInputDialog` method calls in this IO class: one to get the String input and parse it as a double, one to do the same for ints, and one to simply get a String as input. The result is in Listing 6.2 (see next page).

The heading for the `say` method may be puzzling. The `showMessageDialog` permits any displayable Object as its second parameter, not just a String value. You may assign a String value to an Object parameter because String is a subclass of Object (as is every class in Java). You will see the advantage later in this chapter of being able to pass a different kind of displayable Object to the second parameter.

The JOptionPane method that gets a String input returns null if the user clicks the Cancel button. This is usually inconvenient. So the `askLine` method returns the empty String in that case. The entire constructor of the earlier Listing 6.1 could be written with IO more clearly and compactly as follows (compare the two):

Listing 6.2 The IO class

```
import javax.swing.JOptionPane;

public class IO
{
   /** Display a message to the user of Jo's Repair Shop. */

   public static void say (Object message)
   {  JOptionPane.showMessageDialog (null, message,
                "Jo's Repair Shop", JOptionPane.PLAIN_MESSAGE);
   }  //======================


   /** Display the prompt to the user; wait for the user to enter
    *  a string of characters; return that String (not null). */

   public static String askLine (String prompt)
   {  String s = JOptionPane.showInputDialog (prompt);
      if (s == null)
         return "";
      else
         return s;
   }  //======================


   /** Display the prompt to the user; wait for the user to enter
    *  a number; return it, or return -1 if ill-formed.  */

   public static double askDouble (String prompt)
   {  String s = JOptionPane.showInputDialog (prompt);
      return new StringInfo (s).parseDouble (-1);
   }  //======================


   /** Display the prompt to the user; wait for the user to enter
    *  a whole number; return it, or return -1 if ill-formed.  */

   public static int askInt (String prompt)
   {  String s = JOptionPane.showInputDialog (prompt);
      return (int) new StringInfo (s).parseDouble (-1);
   }  //======================
}
```

```
   public GrowthRates()
   {  double rate = IO.askDouble ("Annual rate? 0 if done:");
      while (rate > 0.0)
      {  IO.say ("It takes " + yearsToDouble (rate)
                   + " years for \nyour money to multiply by 2.");
         rate = IO.askDouble ("Another rate (0 when done):");
      }
   }  //======================
```

IO's `askDouble` and `askInt` methods do not use the standard parsing methods that can crash if the numeral is ill-formed. Instead, they use a parsing method developed later in this chapter that returns -1 (or whatever is supplied as the parameter of `parseDouble`) whenever the numeral is ill-formed. The -1 result signals to the calling method that the user made a mistake in keyboard entry when only positive values are expected. This means that the RepairShop software is robust -- as it should be.

The class box for the IO class is in Figure 6.2. In the figure, the words in parentheses describes what kind of values you have to supply to call that method. The word after the colon tells what kind of value the method sends back to you (this is the UML standard notation for the type of value returned by a method, used in a class box; it is not the same notation that Java uses in a method heading).

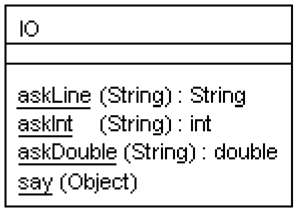


Figure 6.2 The IO class

The conditional operator

The if-else statement in the `askLine` method of Listing 6.2 can be written this way:

```
   return s == null  ?  ""  :  s;
```

The `?:` operator replaces an if-else statement in certain cases: `condition ? x : y` is an expression whose value is `x` if the `condition` is true, but `y` if the `condition` is false. The expressions on each side of the colon `:` must be the same type of value. This book recommends you only use this **conditional operator** for a `return` value or for a value assigned to a variable. If you choose to use it in any other way, you should put parentheses around the entire three-part expression.

In the MathOp class of Listing 5.1, the body of the `average` method can be written as a single statement using the conditional operator:

```
   return sum >= 0  ?  (sum + count / 2) / count
                    :  (sum - count / 2) / count;
```

An independent class method that could be in MathOp to round off a given double value to the closest int value is as follows. The trick is to add 0.5 and see if that makes the number big enough to be rounded up rather than down. For instance, 3.4 rounds to 3 because `3.4+0.5` is 3.9, but 3.6 rounds to 4 because `3.6+0.5` is 4.1:

```
   public static int roundOff (double par)     // in MathOp
   {  return par >= 0  ?  (int) (par + 0.5) : (int) (par - 0.5);
   }  //======================
```

This trick generalizes to round off values to any specific number of decimal places; the following would round off to hundredths (two places right of the decimal point):

```
   return par >= 0  ?  (int) (par * 100 + 0.5) / 100
                    :  (int) (par * 100 - 0.5) / 100;
```

Some examples of how the conditional operator could be used for a value assigned to a variable are as follows:

```
   // assign to lowPay the smaller of hisPay and herPay
   lowPay =   hisPay < herPay  ?  hisPay  :  herPay;

   // assign to halfDay the AM/PM marker for military time
   String halfDay =   military >= 1200  ?  "PM"  :  "AM";
```

The reason for using the conditional operator is not to save space, it is to increase the clarity of the logic. If the two preceding statements were written as if-else statements, it would obscure the fact that the purpose of the statements is to assign a value to `lowPay` and to `halfDay`. With the conditional operator, that purpose is brought to the fore. In any case, any coding written with the conditional operator can be restated without it, so it is not an essential part of the Java language.

Exceptions

A method **throws an Exception object** with a statement similar to the following:

```
   throw new RuntimeException ("Your message here");
```

This notifies the calling method of a problem that crashes the program unless the calling method has special statements that are discussed in Chapter Nine. For instance, if a method contains an int expression involving division, and the divisor has the value zero, that method throws an **ArithmeticException** object. Or if you evaluate `s.length()`, it throws a **NullPointerException** object when `s` has the value null. You should always verify that the Java logic you write can never do this. However, a JOptionPane method with a null value for the prompt-message parameter does not cause problems.

The `Double.parseDouble` and `Integer.parseInt` methods can throw a **NumberFormatException** object if the user enters a badly-formed numeral, such as one with two minus signs or with a comma or letter in it (commas are <u>not</u> allowed within numbers in a Java program). You cannot easily check for this before you call a parsing method. So programs such as those in the earlier Listing 6.1 should have several more statements to handle the Exception object when it is thrown, as described in Chapter Nine, if they are to be used commercially.

Nested classes

One class can be declared inside another class with a heading such as the following:

```
   private static class InsideClass
```

The InsideClass is a **nested class**. The **members of a class** X are the variables, methods, and classes whose declarations are inside X but not inside any member of X.

The effect of putting such a declaration within class X instead of outside class X is <u>only</u> on visibility: Private members of X can be mentioned within the InsideClass; public members of the InsideClass can be mentioned anywhere in X; and no class outside X can mention the InsideClass. The primary purpose therefore is to maintain <u>encapsulation</u> (i.e., so classes outside the "top-level class" X cannot modify variables inside X). This language feature is used in several later chapters.

Language elements
An expression of this form can be used judiciously: Condition ? Expression : Expression
The two values on each side of the colon must be of the same type, which will be the result's type.
You may declare one class inside another with this heading: private static class Whatever
Such a class declaration only affects visibility of identifiers.

Exercise 6.9 Write an IO method `public static int askNonNeg (String prompt)`: It repeatedly asks for an int value until it gets a non-negative value. Use a while-statement.
Exercise 6.10 Rewrite one part of each of Listing 4.5 and Listing 4.6 to use the conditional operator appropriately.
Exercise 6.11* Rewrite parts of Listing 5.5 and Listing 5.10 to use the conditional operator appropriately.
Exercise 6.12* Draw the full UML class diagram for the IO class.
Exercise 6.13** Replace the logic in `askInt` so that it accepts a double value and rounds it down before returning it. Hint: Use `(int)`, but note that `(int)` applied to a negative non-integer rounds up. You need an input of `3.8` returned as `3`, and you need an input of `-3.8` or `-3.2` returned as `-4`.

6.3 Basic String Methods; The Comparable Interface

Java provides the standard library class String in the `java.lang` package. A String object represents a string of characters. You can write a String literal in a program as a string of characters enclosed in quotes. So if you declare the variable `String line`, you can assign `line = "hello"`. That means that the variable `line` refers to an object that contains the five characters `'h'`, `'e'`, `'l'`, `'l'`, and `'o'`, in that order.

Review of String features seen so far

The only operator for String values is the plus sign. Between two String values, the plus sign attaches the second String value to the end of the first; this is **concatenation**. So `"car" + "ted"` is the String value `"carted"`. Between a String value and a numeric value, the plus sign attaches the string of characters that represent the number. So `"car" + 54` is the String value `"car54"`. This plus operator is evaluated left-to-right in the absence of parentheses, so `"car" + 5 + 4` is `"car54"` but `"car" + (5 + 4)` is `"car9"`. The special assignment operator `+=` can be used for Strings, e.g., `s += "it"` means the same as `s = s + "it"`.

You have seen three String methods so far in this book, all of which are illustrated in the following coding to find the word "the" in a sentence:

```
for (int k = 0;  k < sentence.length() - 2;  k++)
{  if ("the".equals (sentence.substring (k, k + 3)))
       return true;
}
```

- `s.length()` is the int value that tells how many characters `s` has.
- `s.equals(t)` is true when `s` and `t` have the same characters in the same order.
- `s.substring(start, end)` is the String value that contains the characters of `s` beginning at position `start` and going up to but not including position `end`. Position numbers are **zero-based**, i.e., the first character is at index 0. This method call throws an **IndexOutOfBoundsException** if `start < 0` or `end > s.length()`. None of Java's String methods change the state of a String value; they at most return new String values. So Strings are **immutable**.

In the for-statement above, the return statement is part of the if-statement. Therefore, that two-line if-statement is the only statement that is subordinate to the for-statement heading. So the braces are not needed. However, coding is probably clearer when you put braces around such compound statements even though not required by the compiler.

STOP

Caution You might think you do not need to test `s.equals(t)` to see whether two Strings are equal; you could just use the `==` operator. But the `==` operator between two object references only tests whether they are exactly the same object. You usually want to know whether they have the same sequence of characters, even though those sequences may be stored in different String objects (i.e., in different places in RAM). If you are ever tempted to use the `==` operator, stop and think: String objects are like boxes, and characters are what are in the boxes, so do you want to know whether two Strings are the same box or whether they have the same contents? General principle: If you want to be safe, never use `==` or `!=` between two object values unless one of them is null.

Also, careless coders may write expressions such as `String s = 47`, trying to put a numeric value in a String variable. Or they may write `return "true"`, which is a String value, in a method that is supposed to return a boolean value. This is like trying to put a square peg in a round hole -- you cannot do it. The compiler will call it an "incompatible type" error.

The backslash character

If you want a quote character to be part of a String literal, you have to write `\"` inside the String literal's quotes. This **backslash character** within a String literal signals to the compiler that the quote character that comes directly after it is a character in the string, not the character that marks the end of the string. For instance, you could print `the game of "Guess My Number"` with the statement:

```
   JOptionPane.showMessageDialog (null,
                "the game of \"Guess My Number\"");
```

In general, Java uses the backslash character in a String literal to signal that the character that follows it has a different meaning from what it would otherwise have. If you want a tab or newline to be part of the String literal, you have to write `\t` or `\n`, respectively. If you want to backspace within the String literal, you have to write `\b`. And if you want the backslash itself in the String literal, you have to write `\\` inside the quotes. These "metacharacters" are often called **escape sequences**.

The compareTo String method

The method call `s.compareTo(t)` compares the two String values `s` and `t`. The number it returns tells which comes earlier alphabetically, if all characters are capital letters or if all are lowercase letters. In other cases, the order of the two String values is determined by a standard set of numerical codes for the characters, as described in the next section (on character values). Technically, it is called **lexicographical ordering**, which is a more general term than "alphabetical ordering."

Figure 6.3 gives details on the `compareTo` method. The numeric result returned by the `compareTo` method is analogous to the value of `n - p` for numbers (since `n - p` is negative if `n < p` and positive if `n > p`).

`s.compareTo (t)`
returns 0 if they are equal, a negative integer if s comes before t, and a positive integer if s comes after t. s and t are Strings.

Figure 6.3 The compareTo String method

Examples `"bid".compareTo("bob")` is a negative integer, and so is `"AL".compareTo("JO")`. So of course `"bob".compareTo("bid")` is a positive integer and so is `"JO".compareTo("AL")`. `"sam".compareTo("sam")` is zero.

As you would expect, if two different Strings `s` and `t` have the same characters up until `s` ends, the shorter one comes before the longer one -- `s.compareTo(t)` is negative. For instance, `"apple".compareTo("applet")` is a negative number.

The **empty String**, i.e., the String with no characters in it, comes before all others in the standard ordering, since it is shorter than all others. The empty String is written in a program as two quote marks `""` right next to each other.

Comparable objects

The String class in the Sun standard library has the phrase `implements Comparable` in its heading. That means that String objects can be passed to method parameters of the Comparable type. The Sun standard library has many classes that implement Comparable, such as the File class and the Date class.

Comparable is not a class, it is a category of classes, called an **interface**. A class is **Comparable** if it has a `compareTo` method with an Object parameter and it returns an int value. The advantage of having the Comparable category of classes is that it promotes reusable software. When you write a method with parameters that are Comparable rather than simply String parameters, you can use that method for Strings, Files, Dates, and other kinds of objects.

Interfaces are used heavily in event-driven programming, discussed in Chapter Ten. For instance, some standard library classes have methods that tell an ActionListener kind of object to respond to the click of a button. If you declare a class that `implements ActionListener`, those library methods tell an object of your class to carry out the actions you choose.

A method with Comparable parameters

The coding of the method in Listing 6.3 produces a message saying whether three given String values are in order. But the method can also be used for File objects, Date objects, and other kinds of objects, because the parameters are of Comparable type rather than String type. The following statement is an example of how it can be called:

```
System.out.println ("The strings are " + ordered (s, t, u));
```

Listing 6.3 An independent class method to test Comparable values

```
/** Tell what kind of order the 3 values are in.
 *  Precondition:  All 3 are mutually comparable. */

public static String ordered (Comparable first,
           Comparable second, Comparable third)
{  if (first.compareTo (second) == 0)
      return (second.compareTo (third) <= 0)
             ?  "in ascending order"  : "in descending order";
   else if (first.compareTo (second) < 0)
      return (second.compareTo (third) > 0)
             ?  "not in order"  :  "in ascending order";
   else  // first comes after second
      return (second.compareTo (third) < 0)
             ?  "not in order"  :  "in descending order";
}  //======================
```

This logic first tests whether the `first` value equals the `second` because if so, all three are in order. Otherwise, the logic splits into two parts, depending on whether the `first` value comes before the `second`. If it does, you need to have the `second` be less than the `third` for ascending order; if it does not, you need to have the `second` value be greater than the `third` for descending order. Note the use of the multiway selection format (`else if` on one line) to avoid excessive indentation and to clarify that there are three possibilities here.

Programming Style The comment after the second `else` in the `ordered` method is intended to make it clear to the reader that, at this point in the logic, `first` is known to come after `second`. There is of course no need to inform the runtime system; it is not sentient. It is good style to avoid testing a condition at a point where you know what the result of the test will be. In particular, it would be pointless to write `else if (first.compareTo (second) > 0)` in place of that commented `else`.

Class casts

You may put `implements Comparable` in the heading of one of your classes if the class has an int-returning method with the signature `compareTo(Object)`. For instance, you could add such a method to the Time class of Listing 4.5, so you could use the `ordered` method and others for Time objects.

The Time instance variables are `itsHour` and `itsMin`. One Time value is "larger" than another if it has a larger value of `itsHour` or, with equal values of `itsHour`, it has a larger value of `itsMin`. So the obvious coding for the `compareTo` method is as follows (because the exact value of the integer returned is not material; only its sign counts):

```
public int compareTo (Object anyTime)  // defective
{  return this.itsHour != anyTime.itsHour
          ? this.itsHour - anyTime.itsHour
          : this.itsMin - anyTime.itsMin;
}  //======================
```

You would of course call the method as `x.compareTo(y)` with two Time variables `x` and `y`. But the method heading has `anyTime` as an Object parameter, not a Time parameter, so the compiler will not let the method body refer to `anyTime.itsHour` or `anyTime.itsMin`. The compiler will complain that Objects do not have instance variables named `itsHour` and `itsMin`. And as usual, the compiler is right.

You cannot fix the problem by making the parameter a Time parameter, because `compareTo` is required to have an Object parameter. What you need is a way to tell the compiler that `anyTime` will be a Time sort of object even though it looks like an Object sort of object. The solution that Java offers is to replace the method definition by the following, in which only the first two lines have been changed:

```
/** Precondition:  ob is a Time object. */
public int compareTo (Object ob)  // in the Time class
{  Time anyTime = (Time) ob;   // promises that ob is a Time
   return this.itsHour != anyTime.itsHour
          ? this.itsHour - anyTime.itsHour
          : this.itsMin - anyTime.itsMin;
}  //======================
```

The first statement of the method uses the **class cast** expression `(Time) ob`, which tells the compiler that at runtime `ob` will refer to a Time object. That allows you to assign it to the Time variable `anyTime`. And that in turn allows you to use `itsHour` and `itsMin` as instance variables. Now the heading of the Time class can be as follows:

```
public class Time extends Object implements Comparable
```

The general principle is that, when you write `(C)x` for some class C, that tells the compiler that `x` will at runtime refer to an object of class C. The compiler accepts such a phrase if C extends the class declared for `x` or implements the interface declared for `x` (either directly or indirectly). But if at runtime the reference in `x` is not to an object of class C or of a subclass of C, a **ClassCastException** is thrown.

By contrast, if you want to assign an object value of class C to a variable of a superclass of C or of an interface that C implements, you do not have to use a cast. The assignment will never throw an Exception. This is in fact what you are doing when you pass `y` to `ob` by the method call `x.compareTo(y)`. And it is what you are doing when you pass the String `s` to `first` by the method call `ordered(s,t,u)`.

The corrected `compareTo` method for the Time class could have used the phrase `((Time) ob)` in three places instead of assigning the value to the `anyTime` variable. But that would execute a bit slower and be harder to read. You would need expressions such as `((Time) ob).itsHour`. The parentheses around the whole expression are needed; the phrase `(Time) ob.itsHour` would be interpreted by the compiler as getting the `itsHour` value from `ob` and then casting that int value to a Time object, which does not make sense.

The calls of `compareTo` in the `ordered` method of Listing 6.3 are polymorphic: If you call `ordered(s,t,u)` for three String values, the runtime system uses the `compareTo` method of the String class. If you call it for three Time values, the runtime system uses the `compareTo` method of the Time class.

The equals method for the Time class

Any class that has a `compareTo` method should also have an `equals` method for which two objects that are equal have a `compareTo` value of zero. A good `equals` method for the Time class would tell whether two different Time objects have the same values for `itsHour` and `itsMin`, with no possibility of throwing an Exception. The following method definition is a reasonable one for this purpose (Section 11.3 gives an even better definition, with an Object parameter, after you learn about some more advanced language features):

```
public boolean equals (Time given)  // in the Time class
{  return given != null
              && this.itsHour == given.itsHour
              && this.itsMin == given.itsMin;
}  //======================
```

Language elements
This following form of expression tells the compiler that the class of the object will be a subclass or an implementation of the expression's type: (ClassName) ObjectExpression
You may append the following to a class heading: implements Comparable
if it has a method with this heading: public int compareTo (Object ob)
A string literal can have \t or \n or \" or \b or \\ within the quotes that start and end it.

Exercise 6.14 What changes would you make in the `ordered` method in Listing 6.3 to print the string of characters within the method instead of returning it?
Exercise 6.15 Write a single statement to print just the number stored in the int variable `x` using `JOptionPane.showMessageDialog`.
Exercise 6.16 Write a statement that prints the substring consisting of the fourth, fifth, and sixth characters in the value stored in the non-null String variable `s`.
Exercise 6.17 (harder) Write an independent method `public static boolean sameFirst5 (String one, String two)`: It tells whether the two given Strings have the same first five characters. Return `false` if either is null or has less than five.
Exercise 6.18 (harder) Write an independent method `public static int indexOf (String big, String little)`: It returns the index in `big` of the first substring of `big` that equals `little`. It returns -1 if `little` is not equal to any substring of `big`. Precondition: Neither of the two String values is null.
Exercise 6.19* Rewrite the `compareTo` method for the Time class without any local variable such as `anyTime`.
Exercise 6.20* Write an `equals` method and a `compareTo` method for the Person class in Listing 5.2, analogous to those for the Time class in this section. One Person object comes before another if it is younger (has a later `itsBirthYear`) or, for Persons with the same birth year, it has a lexicographically earlier `itsLastName`.
Exercise 6.21* Write an application program that reads in three strings of characters from the keyboard and then prints the one that comes first as determined by `compareTo`.

6.4 Character Values And String's CharAt Method

The car repair software will require you to work with individual character values. Java denotes a single character by putting it in apostrophes, as in `'x'` or `'3'` or `'*'`. A variable to store a single character is of **char** type, as in `char letter = 'a'`.

Each char value has a unique whole-number numerical equivalent we call its **Unicode** value. These Unicode values range from 0 to 65,535 inclusive. The main points to remember about the order of the Unicode values are:

- The capital letters are together and in their normal order, that is, `'A'` directly precedes `'B'` which directly precedes `'C'`, etc. through `'Z'`.
- The lowercase letters are together and in their normal order, that is, `'a'` directly precedes `'b'` which directly precedes `'c'`, etc. through `'z'`.
- The digits are together and in their normal order, that is, `'0'` directly precedes `'1'` which directly precedes `'2'`, etc. through `'9'`.
- Lowercase letters come after capital letters which come after digits which come after a blank.

If you combine a char value with an int value using an operator, the system automatically promotes it to the int value corresponding to its Unicode. The same thing happens if you assign a char value to an int variable. You cannot assign an int value or a double value to a char variable without a **cast**, e.g., `chr = (char) ent` is legal when `chr` is a char variable and `ent` is an int value, and so is `ent = chr`, but not `chr = ent`. Also, `chr++` adds 1 to the Unicode of the char value in `chr` and `chr--` subtracts 1.

Two more String methods

Figure 6.4 shows the only other methods in the String class that you need to know for this book. They allow you to find out what the character at a particular position is and to obtain the ending portion of a given String value.

`s.charAt (k)`
is the character at the kth position in the String object referred to by s. An Exception is thrown unless 0 <= k <= s.length() - 1.

`s.substring (n)`
returns a new String object consisting of the characters of s starting at position n and continuing to the end. An Exception is thrown unless 0 <= n <= s.length().

Figure 6.4 Two String methods

Java counts positions in a String value starting from 0, so `"x z".charAt(0)` is the letter `'x'`, `"x z".charAt(1)` is a blank, `"x z".charAt(2)` is the letter `'z'`, and evaluation of `"x z".charAt(3)` throws an **IndexOutOfBoundsException**. Figure 6.5 illustrates the meaning of these String methods. An example of their use is the phrase `word.substring(1) + "-" + word.charAt(0) + "ay"`, making a given word into "PigLatin": That could be used to turn "pot boils" into "ot-pay oils-bay".

Caution If you use the `charAt` method with an index outside the range 0 to `length()-1`, the program throws an IndexOutOfBoundsException. Similarly, `substring(a,b)` requires that `0 <= a <= b <= length()`, otherwise it throws an IndexOutOfBoundsException. Make sure your coding verifies that all index values are within range.

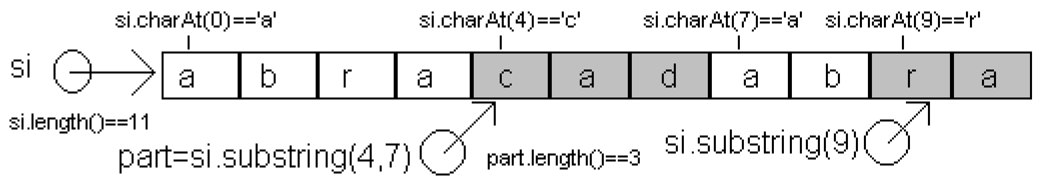


Figure 6.5 Use of String methods

You are perhaps thinking that it is illogical and confusing for characters in a String object to be numbered from 0 on up instead of from 1 on up. But every language has quirks. Just as you won't be speaking correct Spanish unless you learn to use a double negative to actually mean a negative, you won't be writing no correct Java unless you learn to number from 0.

These new String methods allow a more natural rewrite of Listings 5.3 through 5.5. For instance, Listing 5.3 defines `String NONE = "0"`. It could instead define `char NONE = '0'` so NONE could be used as follows, since a plus sign between a char value and a String value concatenates the char onto the string:

```
private static final char NONE = '0';

// replace the last two lines of seesCD by:
   return itsSequence.charAt (itsPos) != NONE;

// replace the body of the if-statement in takeCD by:
   theStack += itsSequence.charAt (itsPos);
   itsSequence = itsSequence.substring (0, itsPos) + NONE
               + itsSequence.substring (itsPos + 1);
```

Class methods versus instance methods

As a general rule, you should strongly prefer to make your methods instance methods rather than class methods. It is easier to complete large software projects successfully if you think in terms of having some object do a job you need to have done (by calling one of its instance methods) rather than doing it yourself (by calling a class method, i.e., with no executor other than you the programmer). But there are several kinds of situations in which a class method is appropriate:

1. You want information about a class of objects as a whole rather than an individual object. For instance, in Listing 5.2, `Person.getNumPersons()` tells how many Person objects have been created.
2. You want information about numbers or characters. For instance, if you want the number 2 to the power `x`, where x is an int value, you cannot use `x.powerOf2()`, because numbers cannot be executors. Thus you have MathOp in Listing 5.1.
3. The main method of an application program is required to be a class method.

Algorithms involving Strings

The String class in the Sun standard library has many useful methods, but sometimes you need more than it offers. For instance, it is useful to be able to determine whether all characters in a given String are between two given characters `lo` and `hi`. Then the method call `someString.inRange('0', '9')` could test whether `someString` has nothing but digits. However, the String class does not have such a method.

The StringInfo class presented next has several useful methods needed for the car repair software, and you may add more if you wish. For instance, if `info` is a StringInfo value, the following statement would see whether it is either all lowercase letters or all capitals:

```
if (info.inRange ('A', 'Z') || info.inRange ('a', 'z'))
   System.out.println (info + " is acceptable.");
```

A **StringInfo** object is basically a wrapper around a String value -- it has one instance variable of type String, named `itself`. This is the use of **composition** rather than inheritance. The implementation of `inRange` requires the common **All-A-are-B looping action**: You look at each character of the String one at a time, from left to right. If you see a character that is not in the specified range, you return `false`. If you get to the end without seeing such a character, you return `true`:

```
public boolean inRange (char lo, char hi)  // for StringInfo
{  for (int k = 0;  k < itself.length();  k++)
   {  if (itself.charAt (k) < lo || itself.charAt (k) > hi)
         return false;
   }
   return true;
}  //=====================
```

By contrast, if you wanted a method to tell you whether <u>at least</u> one character in `itself` is in the range from `lo` to `hi` inclusive, the logic would be the **Some-A-are-B looping action**: Return `true` if you see any character in the specified range; return `false` if you get to the end of the String without seeing one. This gives you the following coding (which you should compare closely with `inRange` above):

```
for (int k = 0;  k < itself.length();  k++)
{  if (itself.charAt (k) >= lo && itself.charAt (k) <= hi)
      return true;
}
return false;
```

The upper part of Listing 6.4 (see next page) contains the constructor, `toString` method, and `compareTo` method for the StringInfo class. The standard name for a method that gives a String equivalent of an object is `toString` (as you saw for the Time class in Chapter Four). The `compareTo` method, which justifies `implements Comparable` in the class heading, requires that you cast its parameter to a StringInfo so you can access `itself`.

The trimFront, firstWord, and retainDigits methods

The RepairShop software will read in a line containing several words, and you will need to be able to separate out the individual words. The `trimFront` method in the middle part of Listing 6.4 is useful for this purpose. It discards the first few characters plus any additional **whitespace** (blanks, tabs, page breaks, or other characters whose Unicode value is less than that of a blank). A parameter gives the starting index. You then increment that index until you come to something that is not whitespace.

Once you have a string of characters with no leading blanks, you need logic to move down the string until you see a blank, then return the characters up to that blank. This is the `firstWord` method in Listing 6.4. An example of how these two methods might be used to strip off the first word of a given StringInfo value named `info` is as follows:

```
info.trimFront (0);
String nextWord = info.firstWord();
info.trimFront (nextWord.length());
```

Listing 6.4 The StringInfo class of objects

```
public class StringInfo extends Object implements Comparable
{
   private String itself;


   public StringInfo (String given)
   {  itself =   (given == null)  ?  ""  :  given;
   }  //======================

   public String toString()
   {  return itself;
   }  //======================

   /** The standard Comparable method.
    *  Precondition: ob is a StringInfo object. */

   public int compareTo (Object ob)
   {  return itself.compareTo (((StringInfo) ob).itself);
   }  //======================

   /** Remove all characters before index n, plus any
    *  whitespace immediately following those characters.  */

   public void trimFront (int n)
   {  while (n < itself.length() && itself.charAt (n) <= ' ')
         n++;
      itself = n > itself.length()  ?  "" : itself.substring (n);
   }  //======================

   /** Return the first word of the String, down to but not
    *  including the first whitespace character.  */

   public String firstWord()
   {  for (int k = 0;  k < itself.length();  k++)
      {  if (itself.charAt (k) <= ' ')
            return itself.substring (0, k);
      }
      return itself;  // since the string has no whitespace
   }  //======================

   /** Strip out all non-digits from the StringInfo object. */

   public void retainDigits()
   {  String result = "";
      for (int k = 0;  k < itself.length();  k++)
      {  if (itself.charAt(k) >= '0' && itself.charAt(k) <= '9')
            result += itself.charAt (k);
      }
      itself = result;
   }  //======================
}
```

You may also need a method that discards all extra spaces, commas, and dollar signs from a numeral, leaving only the digits in the string. The `retainDigits` method in the lower part of Listing 6.4 finds only the digits in the StringInfo executor. It initializes a string `result` to have no characters, then adds any digits it sees to `result`.

The last three methods in this listing use the **count-controlled loop pattern**: Execute the body of the loop a number of times that is known at the time the loop begins, except you may terminate early if you find what you are looking for. That known number of times is `itself.length()` in each method. Note: The standard library StringBuffer class described at the end of Chapter Seven describes how to construct a String of characters more efficiently than is done in the `retainDigits` method.

The `firstWord` method is an example of **sequential search**: You go through a sequence of values to see if you can find a particular value. In this case, you are searching for a blank.

Parsing a string to obtain a number

The IO methods `askInt` and `askDouble` in the earlier Listing 6.2 do not use the Sun standard library `Integer.parseInt` or `Double.parseDouble` methods directly on input from the user. We prefer to avoid an Exception on bad input. The body of the `askDouble` method of the IO class is the following:

```
   String s = JOptionPane.showInputDialog (prompt);
   return new StringInfo (s).parseDouble (-1);
```

This method call returns the double value that the first part of the string represents, even if there are some unacceptable characters after that acceptable first part. For instance, if the string is "53x" or "-53,217" or ".24 ", it returns 53 or -53 or 0.24 respectively. If the initial part of the string does not form a valid numeral, this `parseDouble` method call returns -1, the parameter value. This special return value allows you to check whether the conversion from a numeral to a number succeeded, since the RepairShop software does not expect a negative number for input. The body of IO's `askInt` method is the same as its `askDouble` method except for an `(int)` cast before the return value.

Listing 6.5 (see next page) has StringInfo's `parseDouble` method. The first step is to return the parameter value if the string has no characters. Otherwise you set `pos` to the first index that might be a digit (index 1 if the first character is a minus sign, otherwise index 0). Then you advance `pos` past any digits that occur. If the first nondigit is a decimal point, you advance `pos` past any digits that occur after the decimal point. As you advance, you make a note of whether you saw at least one digit either before or after the decimal point (since both "44." and ".44" are acceptable numerals). If you saw at least one digit, you can then safely call `Double.parseDouble` on the substring up to the first invalid character, otherwise you return the parameter value.

Language elements
A char value represents a single character. Each character has a Unicode value 0 to 65,535.
If you assign a char value to a numeric variable or use a numeric operator with it, it is converted to the int value corresponding to its Unicode. Assigning a double or int value to a char variable requires you put the cast operator `(char)` in front of the numeric expression. Technical note: The cast can be omitted for int constants 0 to 65,535, but it is better to use the cast all the time.

Exercise 6.22 If `s` is the String value "algorithm", what is `s.charAt(2)`? What is `s.substring(2)`? What is `s.substring(2, 6)`?
Exercise 6.23 The `firstWord` method is only supposed to be called when the given String does not begin with whitespace. What happens if it does begin with whitespace?
Exercise 6.24 Write a StringInfo method `public int countRange (char lo, char hi)`: the executor tells how many characters in itself are in the range `lo` to `hi` inclusive. For instance, `info.countRange('A','Z')` counts all capital letters.
Exercise 6.25 Write a StringInfo method `public char biggestChar()`: The executor returns the character in itself that has the highest Unicode value. If the string is empty, return `(char) 0`.

Listing 6.5 The parseDouble method in the StringInfo class

```java
   /** Return the numeric equivalent of itself.
    *  Ignore the first invalid character and anything after it.
    *  If the part of the string before the first invalid
    *  character is a numeral, return the double equivalent,
    *  otherwise return the value of badNumeral. */

   public double parseDouble (double badNumeral)
   {  if (itself.length() == 0)                              //1
         return badNumeral;                                  //2
      boolean hasNoDigit = true;  // until a digit is seen   //3
      int pos =  (itself.charAt (0) == '-')  ?  1  :  0;     //4
      for ( ;  hasDigitAt (pos);  pos++)                     //5
         hasNoDigit = false;                                 //6
      if (pos < itself.length() && itself.charAt(pos) == '.')//7
      {  for (pos++;  hasDigitAt (pos);  pos++)              //8
            hasNoDigit = false;                              //9
      }                                                      //10
      return hasNoDigit  ?  badNumeral                       //11
             :  Double.parseDouble (itself.substring (0, pos));
   }  //======================

   private boolean hasDigitAt (int pos)
   {  return pos < itself.length() && itself.charAt (pos) >= '0'
                                   && itself.charAt (pos) <= '9';
   }  //======================
```

Exercise 6.26 (harder) Write a StringInfo method `public void trimRear()`: The executor discards all the whitespace at the end of itself.

Exercise 6.27 (harder) Write a StringInfo method `public boolean hasWhite()`: The executor tells whether it contains at least one character less than or equal to a blank.

Exercise 6.28 (harder) Write a StringInfo method `public int indexOf (char par)`: The executor returns the position number of a given character in itself; it returns -1 if the character is not there. In case the character is in itself several times, it returns the position of the first instance of that character.

Exercise 6.29 (harder) Write a StringInfo method `public String initDigits()`: The executor returns the initial portion of itself that consists of digits (up to the first non-digit).

Exercise 6.30* Write a StringInfo method `public String signedNumber()` that does what the preceding exercise does except that it includes an initial negative sign if it is directly followed by a digit. Call the `initialDigits` method as part of the logic.

Exercise 6.31* Revise the method of the preceding exercise to allow at most nine digits, or ten if the first digit is a 1 (due to the limits on the size of an int value).

Exercise 6.32* Write a StringInfo method `public int lastBiggie (StringInfo par)`: The executor returns the last position number at which it has a character that has a larger Unicode than the character of the parameter at the same position number (-1 if there is no such character). It throws an Exception if `par` is null.

Exercise 6.33** Write a StringInfo method `public StringInfo reverse()`: The executor returns a new StringInfo value of the same length but with the characters in reverse order. Hint: You cannot assign a new value at a particular index in a string, but you can concatenate characters on to a string one at a time to get what you want.

Exercise 6.34** Write a StringInfo method `public void replace (char lo, char hi)`: The executor replaces every instance of the char value `lo` by the char value `hi` and makes no other change in itself.

Exercise 6.35** Write a StringInfo method `public void numWords()`: The executor tells how many words it has (a word is non-whitespace followed by either whitespace or the end of the string).

6.5 Long Integers; Casts And Conversions; The Math Class

You may declare a variable as type **long**. That means that it can store an integer value with up to 63 binary digits, in addition to a negative sign if needed. So the range of values is plus or minus slightly over eight billion billion, which is a 19-digit number.

One advantage of long values is that you can store money amounts up to 80 million billion dollars, exactly to the penny. An int value has 31 binary digits plus a possible negative sign, so an int variable cannot store more than about 20 million dollars exactly to the penny. The only disadvantages of using long values versus int values are that they take twice the space in RAM and operations with long values take at least twice as long.

Reading the system clock

The following statement uses a Sun standard library method that returns a long value:

```
   long rightNow = System.currentTimeMillis();
```

It gives the current time, measured in the number of milliseconds that have passed since midnight of January 1, 1970. There are 1000 milliseconds in a second, and 86,400 seconds in a day (multiplying 60*60*24), for a total of 86,400,000 milliseconds in a day. So if the current time were returned as an int value, it would give the wrong answer after a little less than a month. That is why the current time is returned as a long value.

If you have a method call `doStuff(x)` that works with an Object x, and if you want to know how much time it takes to execute that method call, you could call the following method. It checks the time just before the method executes, then again after it finishes execution, and reports the difference in the two times.

```
   public static int getElapsedTime (Object x)
   {  long start = System.currentTimeMillis();
      doStuff (x);
      return (int) (System.currentTimeMillis() - start);
   }  //======================
```

Conversions

The char, int, long, double, and boolean types are called **primitive types** of values, since they are not objects. Java has three other primitive types, all numeric. These are discussed in Chapter Eleven, but they are not used in this book.

You may assign a long value to a double variable without an explicit cast. But you cannot assign a long value to an int or char variable without an explicit cast, as in `intVariable = (int) longVariable`. The "narrow-to-wide" hierarchy

```
   char -> int -> long -> double
```

summarizes casting requirements: You must cast a value of one of these types to assign it to a variable of a type further to the left. You do not need to cast if you assign a value to a variable of a type further to the right. Note that boolean is not in this hierarchy; you cannot cast a boolean value to or from any other primitive type of value.

One **byte** of space is 8 bits, corresponding to 8 base-two digits. So you may store up to 256 (2^8) different values in one byte of space. Unicode values (for chars) require two bytes of storage space, since they range from 0 up to 65,535 (2^{16}). int values require four bytes, since they range up to plus or minus 2^{31}. long and double values require eight bytes (long values range up to plus or minus 2^{63}).

Using a value of char or int type as if it were "wider" (by combining it with a "wider" number using an operator or assigning it to a "wider" variable) is a **widening conversion**, or **promotion**, and does not lose information (except a little bit with longs to doubles). So the compiler does this automatically. Assigning to a "smaller" variable (e.g., double to long or long to int or int to char) is a **narrowing conversion**. The compiler requires you to acknowledge the possibility of a loss of information by using a cast for a narrowing conversion. Combining a value with a "wider" value is called "arithmetic conversion"; assigning it to a "wider" variable is called "assignment conversion".

This is analogous to the rule for class casts (discussed in Section 6.3): If Sub is a subclass of Super, and you have declared `Sub x; Super y;` then you may make the assignment `y = x` but assigning the other way requires a cast: `x = (Sub) y`. The latter is analogous to narrowing, since `x` is more specific than `y`; the `y = x` assignment is analogous to widening.

Note: Subtracting an int value from a char value converts the character to its Unicode value. If `ch` is a char variable that contains a capital letter, then `(ch - 'A')` will be an int value in the range from 0 to 25. If `ch` contains a lowercase letter, then `(ch - 'a')` will be in the range from 0 to 25. And if `ch` contains a digit, then `(ch - '0')` will be in the range from 0 to 9 (inclusive in each case).

The Math class

The Sun standard library has a class of methods that perform some basic calculations. Some of the methods in this **Math** class are as follows. Each of these methods is a class method, as you can see from the fact that the class name appears in place of a reference to the executor:

- `Math.sqrt(x)` returns the square root of x, e.g., `Math.sqrt(25)` is 5. If you try to calculate the square root of a negative number, it returns the value **NaN**, which stands for "not a number".
- `Math.abs(x)` returns the absolute value of x, e.g., `Math.abs(-12)` is 12.
- `Math.min(x, y)` is the smaller of x and y, e.g., `Math.min(5,-12)` is -12.
- `Math.max(x, y)` is the larger of x and y, e.g., `Math.max(5,-12)` is 5.
- `Math.pow(x, y)` returns x-to-the-power-y, e.g., `Math.pow(2,5)` is 32.
- `Math.random()` returns a "random" number `r` such that `0 <= r < 1`.
- `Math.cos(x)` returns the cosine of x.
- `Math.sin(x)` returns the sine of x. Other trigonometric functions are available.
- `Math.log(x)` is the natural logarithm of x, that is, to base e. If you try to calculate the logarithm of a negative number, it returns the value NaN.
- `Math.exp(x)` is e to the power x, so `Math.exp(Math.log(x))` is x.

Each of these Math methods returns a double value calculated from one or two double parameters, except that `abs`, `min`, and `max` will instead return an int value when the parameter(s) are int values, or a long value when the parameters are long. The Math class also contains two constants: `Math.PI` is the area of a circle of radius 1, roughly 3.14159, and `Math.E` is the natural base of logarithms, roughly 2.71828.

Crash-guards

The following three if-statements have something in common. Do you see what it is?

```
if (sue != null)        s = sue.getName();
if (k < par.length())   s = "" + par.charAt (k);
if (x != 0)             y = 6 / x;
```

The subordinate statement in each case would throw an Exception if it were executed when the condition is false, e.g., `sue.getName()` throws a **NullPointerException** if `sue` contains the null value. But the if-statement guards against such a "crash". The following three conditions also have something in common. Do you see what it is?

```
sue != null && sue.getName().equals ("bob")
k < par.length() && par.charAt (k) > ' '
x != 0 && 6 / x < 3
```

The part after the and-operator in each case would throw an Exception if it were evaluated when the part before the and-operator is false. But the short-circuit property of the and-operator guards against such a crash.

In each case, the condition at the beginning of the phrase is a **crash-guard** for the expression or statement in the latter part of the phrase. A crash-guard prevents the evaluation of an expression or statement in a case when the expression or statement is impossible to evaluate sensibly.

A crash-guard can also be a condition that, when true, prevents the evaluation of a phrase that would throw an Exception. This happens when the guarded part is subordinate to `else` or comes after a conditional return, as follows:

```
if (x == 0)  { /* misc. statements */ }  else  y = 6 / x;
if (x == 0)  return 0;  y = 6 / x;
```

And of course, the short-circuit property of the or-operator can be used to crash-guard an expression or statement, as in the following conditions:

```
k >= par.length()  ||  par.charAt (k) <= ' '
x == 0  ||  6 / x < 3
```

Programming Style As a general principle, you should crash-guard all expressions and statements that could possibly throw an Exception under certain circumstances, when those circumstances can be guarded against with one or two simple tests of conditions. However, you do not need to guard against circumstances that are expressly prohibited by the precondition of a method. The precondition states what crash-guarding has already been done before the method is called, so further crash-guarding would be superfluous.

Language elements
The narrow-to-wide hierarchy char => int => long => double for primitives determines casting: You must use a cast to have a value treated as a "narrower" value, but not to have a value treated as wider. A long integer value ranges up to plus or minus 2^{63} - 1.

Exercise 6.36 Write an independent method `public static char toUpperCase (char par)`: It returns the capitalized form of the char parameter. It returns the parameter without change if the parameter is not lowercase.
Exercise 6.37 Write an independent method `public static double largestOf4 (double x, double y, double z, double w)`: it returns the largest of four double parameters. Have just one statement. Use `Math.max` three times.
Exercise 6.38 Write an application program that reads in a double value and prints its square root, fourth root, and eighth root. Assume numeric input is positive well-formed.
Exercise 6.39 State which of the expressions or statements in Listing 2.8 has a crash-guard, and state which of the four types of crash-guard it is.
Exercise 6.40* Same question as the previous exercise, but for Listing 3.10.
Exercise 6.41* Same question as the previous exercise, but for Listing 5.5.
Exercise 6.42* Java has its own analog of the Y2K bug, in that times and dates will be messed up when the current time exceeds the capacity of a long value. In approximately what year will that happen?

Part B Enrichment And Reinforcement

6.6 Formatted Output To A JTextArea In A JScrollPane

The car repair software requires the display of a large amount of output in a graphical window. `IO.say` is not appropriate for this part of the output, since you will print some of the data at one point in the program, then more at another point, etc. Also, the user may want to review the output from previous steps. You could use `System.out.println`, but the alternative presented in this section is often preferable.

The JTextArea class

You may construct a rectangular area for the output of information with the following statement:

```
   JTextArea area = new JTextArea (10, 25);
```

The area is big enough for 10 lines (the first parameter) of about 25 characters each (the second parameter). That is, the parameters tell the height and width in that order. **JTextArea** is in the `javax.swing` package. The text area has a text string that it displays, initially of length zero. You may then add a string of characters to the information that appears in the text area with an `append` statement:

```
   area.append ("testing");
   area.append ("\n a JTextArea");
```

The text area will then have two lines to display, "testing" and "a JTextArea". The following statement displays the text area in a JOptionPane window:

```
   JOptionPane.showMessageDialog (null, area);
```

Note that you may supply a JTextArea object in place of the String object you have previously used. All that is required is that the second parameter of `showMessageDialog` be a displayable Object. You may then append a few more lines, so that the next time you execute the `showMessageDialog` statement, all the lines appear, including those that appeared earlier. But if you want to start fresh with a new string of characters, use something like `area.setText("new stuff")`.

The JScrollPane class

The user will want to be able to scroll up and down in a JTextArea when the amount of information becomes quite large. You may add a scrollbar to the JTextArea object named `area` with the following statement:

```
   JScrollPane scroll = new JScrollPane (area);
```

JScrollPane is in the `javax.swing` package. The JScrollPane constructor takes a JTextArea object as its parameter so the scrollbar object knows what to wrap itself around. When you are ready to display it, you use the following statement:

```
   JOptionPane.showMessageDialog (null, scroll);
```

Example of using a text area

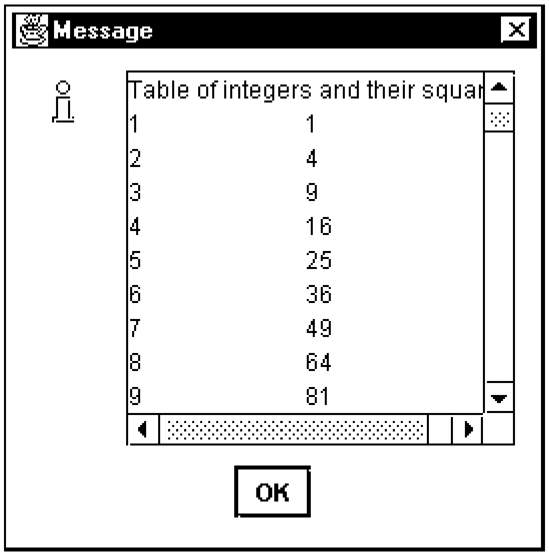
Listing 6.6 illustrates the use of these standard library methods. It creates a scrolled text area 10 characters tall and 15 'm' characters wide (i.e., using the width of the letter 'm' to measure). It then prints all of the integers from 1 to 100, each with its square one tab position to the right (the `"\t"` part makes the tab character). Note that `import javax.swing.*;` saves writing the phrase `javax.swing` in five different places.

Listing 6.6 The PrintSquares object class, with a test application

```
import javax.swing.*;

public class PrintSquares
{
   /** Print a table of integers 1..10 and their squares. */

   public PrintSquares()
   {  JTextArea area = new JTextArea (10, 15);                //1
      JScrollPane scroller = new JScrollPane (area);          //2
      String table = "Table of integers and their squares";  //3
      for (int k = 1;  k <= 100;  k++)                        //4
         table += "\n" + k + "\t" + k * k;                    //5
      area.append (table);                                    //6
      JOptionPane.showMessageDialog (null, scroller);         //7
   }  //======================
}  //########################################################


class PrintSquaresTester
{
   public static void main (String[ ] args)
   {  new PrintSquares();                                     //8
      System.exit (0);                                        //9
   }  //======================
}
```

Figure 6.6 shows what the dialog box could look like at the end of execution of PrintSquares. The scroll bar on the right indicates that it displays only about 10% of the total output. The width was deliberately chosen to be somewhat smaller than needed so you can see where it cuts off (at about 32 smaller characters).

You have now seen two ways a program can send data outside itself: `System.out.println` for terminal window and `showMessageDialog` for modal graphic. The dialog box method is called **modal** because the program pauses until the user clicks a response.



```
Table of integers and their squar
1            1
2            4
3            9
4            16
5            25
6            36
7            49
8            64
9            81
```

Figure 6.6 Screen shot of JTextArea

The text area and the scroll pane work together to accomplish a single task, displaying to the user a view of the data produced by the program. So these two objects should be logically represented as a single object, which we call a View object. The RepairShop software will require such an object, as defined in Listing 6.7.

Listing 6.7 The View class of objects

```
import javax.swing.*;

public class View extends Object
{
   private JTextArea area;
   private JScrollPane scroller;


   public View (int rows, int columns)
   {  area = (rows < 1 || columns < 1)  ?  new JTextArea (2, 10)
                                        :  new JTextArea (rows, columns);
      scroller = new JScrollPane (area);
   }  //======================


   /** Show the message in a scrolled text area of a dialog. */

   public void display (String message)
   {  area.append (message); // no effect if message == null
      IO.say (scroller);
   }  //======================
}
```

The View constructor has a crash-guard in case of non-positive int values. The `display` method does not need a crash-guard against a null String value or an empty String, since the specifications for the `append` method in JTextArea say that nothing happens in such a case. This class definition lets you replace lines 1 and 2 in the earlier Listing 6.6 by

```
   View output = new View (10,15);
```

and replace lines 6 and 7 by

```
   output.display (table);.
```

Exercise 6.43* Revise Listing 6.1 to use the View class for all output.
Exercise 6.44** Revise the Mastermind game in Listing 4.9 to display all output so far each time it shows the updated status, as well as remind the user of the number chosen.

6.7 Analysis And Design Example: The RepairOrder Class

Your initial analysis of the RepairShop software showed the need for decimal numbers and for objects that could handle sophisticated input and output, which you now have. It also showed the need for objects that represent work orders. So the next task is to create a RepairOrder class.

First you consider what kind of behavior a work order must have. You should be able to ask a work order object for the client's name. You should also be able to ask it for the kind of repair work to be done. The foreman will then look at the job to make an estimate of how much time the repair will take. So you need to be able to tell a work order to remember that estimate so you can retrieve it later. Finally, it will be useful to be able to tell a work order that has been completed to send a bill to the client.

Now that you know the behavior that a RepairOrder object should exhibit, you can see what knowledge a RepairOrder object needs, that is, its instance variables. You should store the client's first and last name and the repair work that has to be done. That is three String values. These values can be determined at the time the work order is first created. The estimate of the time can be a fractional number of hours (e.g., 1.75 for 1 hour and 45 minutes), so a double variable is suitable for storing that value.

Most object classes should have an instance method that supplies all or most of the information stored in an object of the class in a printable form. Such a method often comes in handy. The name `toString` is the conventional Java name for the String analog of an object.

Stubbed documentation

This leads you to develop documentation for the RepairOrder class as the sketch shown in Listing 6.8 (see next page). The methods in this documentation are **stubbed**. That is, the body of each has the minimum necessary to make it compile -- nothing if a void method, a simple return statement otherwise (null if it returns an object).

A stubbed class such as this is just for discussion and development purposes. The only reasons for making it compilable are (a) it is a format understood by all Java programmers who might be in on the discussion, and (b) there is something satisfying about having it compile.

Sometimes you put something more in stubbed methods, such as `IO.say ("now in method XXX")`. This would be because you want to actually run a program that calls on those methods, but you are not yet ready to develop their logic. All you would be doing is testing out some other methods, perhaps seeing how the user interface looks, that require that these exist.

Figure 6.7 shows how a typical RepairOrder object is represented. The RepairOrder variable named `ralph` is an object reference, so its value is shown as an arrow pointing to a rectangle with no name (because objects themselves are not named variables). That rectangle contains four variables, three of which contain String references. Since Strings are objects, their values are also shown as arrows pointing to rectangles.

Listing 6.8 Documentation for the RepairOrder class

```
public class RepairOrder extends Object  // stubbed documentation
{
   private String itsFirstName;
   private String itsLastName;
   private String itsRepairJob;
   private double itsEstimatedTime;


   /** Create a RepairOrder from a line of input.  */
   public RepairOrder (String par)                          { }

   /** Return the client who owns the car. */
   public String getClient()                    { return null; }

   /** Return the kind of repair job to be performed. */
   public String getRepairJob()                 { return null; }

   /** Return the estimate of the time to do the job. */
   public double getEstimatedTime()              { return 0.0; }

   /** Record the estimate of the time to do the job. */
   public void setEstimatedTime (double time)               { }

   /** Send a bill to the client. */
   public void sendBillToClient()                           { }

   /** Return a String value containing most or all of the
    *  RepairOrder's data, suitable for printing in a report. */
   public String toString()                     { return null; }
}
```

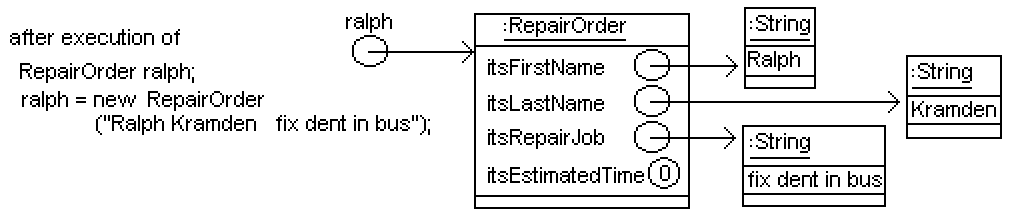


Figure 6.7 UML object diagram for a RepairOrder variable

A reasonable definition of the RepairOrder class is in Listing 6.9 (see next page). The constructor and the `sendBillToClient` methods are the only ones that cannot be done in a single statement. We will skip the `sendBillToClient` method in this development, since we do not need it for the key logic of scheduling repair jobs.

The `toString` form of an object typically gives the value of most or all of the instance variables of the object. It is reasonable for RepairOrder objects to supply all four of these attributes, separated by punctuation and blanks (in the lower part of Listing 6.9).

Development of the RepairOrder constructor

To develop the logic for the constructor, you first need to make sure that you have not been passed no String at all (i.e, the null value); if you have, you leave the instance variables with their initial values.

Listing 6.9 The RepairOrder class

```
public class RepairOrder extends Object
{
   private String itsFirstName = "";
   private String itsLastName = "";
   private String itsRepairJob = "";
   private double itsEstimatedTime = 0.00;


   public RepairOrder (String par)
   {  if (par != null)
      {  StringInfo si = new StringInfo (par);
         si.trimFront (0);  // remove any leading whitespace
         this.itsFirstName = si.firstWord();

         si.trimFront (this.itsFirstName.length());
         this.itsLastName = si.firstWord();

         si.trimFront (this.itsLastName.length());
         this.itsRepairJob = si.toString();
      }
   }  //======================


   public String getClient()
   {  return itsFirstName + " " + itsLastName;
   }  //======================


   public String getRepairJob()
   {  return itsRepairJob;
   }  //======================


   public double getEstimatedTime()
   {  return itsEstimatedTime;
   }  //======================


   public void setEstimatedTime (double time)
   {  if (time > 0.0 && time < 20.0)
         itsEstimatedTime = time;
   }  //======================


   public String toString()
   {  return itsLastName + ", " + itsFirstName + ": "
             + itsRepairJob + ", time= " + itsEstimatedTime;
   }  //======================
}
```

If the String parameter is not null, you need to strip off any initial blanks (and other whitespace). Then you get the first word and store it as the first name of the client for this repair job. You should use a StringInfo object to do this (as defined in the earlier Listing 6.4). Then you can use the `trimFront` and `firstWord` instance methods from that class.

What remains is the part of the String parameter that comes after that first word; you can find that easily by taking the substring of `par` that begins with the character after the first word. You repeat the process to get the second word, which is the client's last name. Whatever is left must be the description of the repair job.

You may be thinking that the RepairOrder constructor could produce some kind of error if the given string of characters does not have any non-blank characters. However, there is really no problem in this case. After the first name is removed, the string that is left is the empty string, `trimFront` will leave it the empty string, and then `firstWord` applied to the empty string will return the empty string. The body of the looping statement in each of those two methods is not executed at all, since `itself.length()` is zero.

Exercise 6.45 Write a `setEstimatedTime` RepairOrder method with no parameter: The executor asks the user for an estimated time and updates itself accordingly, but only if its current estimated time is zero. Assume that numeric input is well-formed.
Exercise 6.46 (harder) What changes should be made in Listing 6.9 to give each RepairOrder object a unique ID, which is a single capital letter? Provide a method for retrieving the ID of a RepairOrder object. Assume that no more than 26 RepairOrders will ever be created.
Exercise 6.47* What changes would you make in Listing 6.9 to add an instance variable `itsEstimatedCost` that is obtained analogously to `itsEstimatedTime`?
Exercise 6.48* Add a class variable to RepairOrder that keeps track of the total of the estimated times of all RepairOrder objects that have been created. Also add a class method that allows outside classes to access that total estimated time.
Exercise 6.49* Draw the UML class diagram for the RepairOrder class.

6.8 Analysis And Design Example: Model/View/Controller

The software that schedules the repair jobs needs to maintain a queue that contains all the repair jobs left to do. A queue is just a list of things for which the primary operations are adding a new value and removing an old value; the latter operation always removes the value that has been on the list for the longest period of time.

Analysis

You talk with the client to clear up some details of how the RepairShop software is to behave. Question How many jobs do you allow space for? Answer The client tells you that it could be anywhere from 5 to 20 jobs in one day, so you decide to make sure the queue has room for at least 100 (just to be safe).

Question How does the client want to indicate whether the next operation to process is (a) to enter a new repair job or (b) to remove an existing repair job from the queue for processing? Answer The client tells you that it will be convenient to enter `"A"` to indicate adding a new job.

Since clients are often not sensitive to the fact that capital letters are not interchangeable with lowercase letters in a Java program, you check back and verify that either `"A"` or `"a"` is to indicate adding a new job. Also, either `"X"` or `"x"` is to indicate terminating the program.

The standard library String class has a useful instance method named `toUpperCase` that returns a new String with every lowercase character replaced by its uppercase equivalent. So we use the following phrase in our coding so that `"A"` and `"a"` are treated the same (we do not use this special String method elsewhere in this book):

```
IO.askLine (CHOICES).toUpperCase();
```

Main logic Design

This all leads to the design of the main logic shown in the accompanying design block, after extensive thought about how to go about it.

STRUCTURED NATURAL LANGUAGE DESIGN for the main logic, first draft

```
1.  Create a queue named jobQueue that can hold up to 100 jobs (far more than might
    be required).
2.  Ask the user for the first repair job and add it to the jobQueue.
3.  Repeat the following until the user decides to quit...
        3a.  If the user wants to add a new job at this point then...
                3aa.  Ask the user for that repair job and add it to the jobQueue.
        3b.  Otherwise, if the jobQueue is not empty...
                3ba.  Remove the next job to do from the jobQueue and display it.
        3c.  Report the total estimated time for all jobs still in the jobQueue.
```

After you develop this design, you study it carefully before going further, to make sure it is right. When you do, you can see a defect: It makes no provision for calculating the total estimated time. You cannot report to the user values that you have not calculated. So you expand the main logic to allow for these calculations. You also did not create an object to display information to the user. The View kind of object from Listing 6.7 is intended for this purpose. So the revised plan is as shown in the larger accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for the main logic, second draft

```
1.  Create a View object to display the information to the user.
2.  Create a queue named jobQueue that can hold up to 100 jobs (far more than might
    be required).
3.  Ask the user for the first repair job and add it to the jobQueue.
4.  Initialize a running total of estimated times to that first job's estimated time.
5.  Repeat the following until the user decides to quit...
        5a.  If the user wants to add a new job at this point then...
                5aa.  Ask the user for that repair job and add it to the jobQueue.
                5ab.  Add to the running total to include that new job.
        5b.  Otherwise, if the jobQueue is not empty...
                5ba.  Remove the next job to do from the jobQueue and display it.
                5bb.  Subtract from the running total to allow for that removed job.
        5c.  Report to the user the running total estimated time for all jobs remaining.
```

Object Design

You already have a description of RepairOrder objects, IO objects, and View objects. Clearly you need a Queue object as well, with the capabilities of adding a value, removing a value, and telling you whether or not it is empty. The Sun standard library does not contain a Queue class. Fortunately a friend of yours has a Queue class handy, thoroughly debugged and tested, that you can use (the full implementation of that Queue class is at the end of the next chapter). Its available operations include the following:

- `new Queue()` creates a Queue object that can hold any number of jobs.
- `jobQueue.enqueue (x)` adds the job `x` to the queue.
- `jobQueue.dequeue()` removes and returns the next available job on a first-in-first-out basis.
- `jobQueue.isEmpty()` tells whether the queue is empty.

The overall **object design** for the RepairShop software is the information in Figure 6.8, together with a description of what services each of these methods provides.

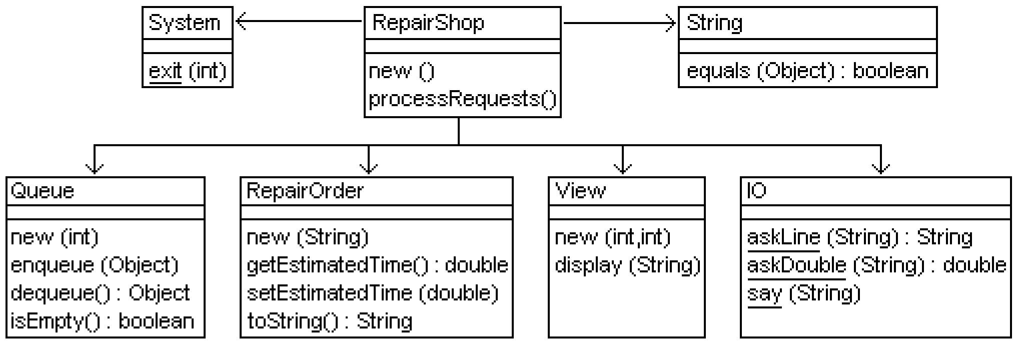


Figure 6.8 UML class diagram for RepairShop

Cowboy hats

Your friend did not know, when she wrote this Queue class, that you would want it for storing RepairOrder objects. It would be of limited value and not reuseable if it only stored RepairOrder objects anyway. She wrote it to store Object objects, which is the most general (and therefore the most useful) kind of object to store. Reminder: The Object class is the superclass of every class.

The only problem with this is that the method call `jobQueue.dequeue()` produces an Object value, and the compiler will not let you store an Object value in a RepairOrder variable (you cannot mix up object types like that). It would be like putting an ordinary person on a rodeo bull when the rules say that only a cowboy can ride a rodeo bull (if you think this is another metaphor coming up, you are mistaken; it is a simile).

If `x` is a reference to an object and ClassName is the name of some class, then the class cast `(ClassName) x` is a reference to an object of type ClassName. It is as if you put a cowboy hat on it so the compiler will think of it as a cowboy.

There is, however, a caveat: If during execution of the program the runtime system evaluates the phrase `(ClassName) x` and finds that `x` is not actually of type ClassName, then it throws a ClassCastException. In other words, not everyone who wears a cowboy hat is a cowboy.

Consequently, `(RepairOrder) jobQueue.dequeue()` produces a RepairOrder value from an Object value. Although this is analogous to the expression `(int) someDouble`, it is substantially different in two ways: The `(int)` cast causes the runtime system to modify the value it gets from `someDouble` by chopping off (truncating) the part after the decimal point; it does not throw an Exception. The class cast does not modify anything, but it causes the runtime system to throw an Exception if the object returned by `jobQueue.dequeue()` is not in fact a RepairOrder object.

The Model/View/Controller pattern

The Java coding for this RepairShop application program follows fairly directly from the previous discussion. Listing 6.10 gives the result (see next page). The constructor handles steps 1 through 4 of the main logic design and the `processRequests` method handles the complex fifth step of the main logic design. This class illustrates the Model/View/Controller pattern as a way of organizing a program:

- One kind of object, both Queue and RepairOrder in this case, stores the data during execution of the program, modifies that data when told, and provides information about the data when asked. This is the **Model** aspect. It provides no input or output.
- Another kind of object, **View** in this case, displays the information about the model for the user to see. In more complex software, there may be several views of the data structure (such as a histogram and a numerical table).
- A third kind of object, RepairShop in this case, serves the **Controller** function. Its job is to accept user input, then send messages to the Model (asking it to update itself or to give information about itself) and to the View (asking it to update the display) as needed.

In Chapter Ten, when you learn about event-driven programming, the Controller function will be fulfilled by the objects that "listen" for the user's click of a button or entry of information in a textfield and respond as described above. This **Model/View/Controller** pattern is quite common in programming that uses such objects. In general, we delegate the responsibilities of the program to three distinct kinds of objects whose purposes are (a) maintaining the model of an aspect of the real world, (b) displaying the information to the user, and (c) obtaining input from the user and reacting to it.

In the typical case, a Controller either tells the View how it changed the model, or it simply informs the View that it changed the Model and lets the View ask the Model how. Either way, the Model is not even "aware" that it participates in an MVC triad.

<u>Programming Style</u> Keep all your methods down to where they will fit comfortably on one screen. Create objects to carry out significant subtasks so you can avoid excessive complexity in your main logic. Define separate classes of objects to do this. In general, you should rarely have other methods in the class with the main method; this book never does.

Exercise 6.50 (harder) Write a second RepairOrder constructor to add to the earlier Listing 6.9: It has two parameters, the string that describes the repair order plus a double value that specifies the estimated time for that job. Revise Listing 6.10 to use this constructor. Precondition: Neither parameter is null.
Exercise 6.51* Rewrite the first if-condition in Listing 6.10 to use `charAt`. Allow for the empty String.
Exercise 6.52* Revise Listing 6.10 so that the View prints the number of jobs still on the queue. Extra credit: Have a new kind of object keep track of the `totalTime` and the number of jobs, as well as of the queue itself (a smarter kind of queue will do).
Exercise 6.53* Revise Listing 6.10 so that it forces the user to enter again each numeric input that is not a valid numeral or is negative.

Listing 6.10 The RepairShop object class, with a tester application

```java
public class RepairShop
{
   /** Repeatedly get repair jobs from the foreman or display
    *  the next repair job to the foreman, until there are no
    *  more jobs to be done. */

   private static final String ENTER
         = "Enter first name, last name, and description of job";
   private static final String TIMEPROMPT
         = "Estimate the time to complete the repair job";
   private static final String CHOICES = "Enter A to add another"
         + " job, X to exit, anything else to process a job";
   //////////////////////////////////////////////////
   private View output = new View (40, 25);
   private Queue jobQueue = new Queue();
   private RepairOrder nextJob = new RepairOrder
                                    (IO.askLine (ENTER));
   private double totalTime = IO.askDouble (TIMEPROMPT);


   public RepairShop()
   {  nextJob.setEstimatedTime (totalTime);                  //1
      jobQueue.enqueue (nextJob);                            //2
   }  //======================


   public void processRequests()
   {  String input = IO.askLine (CHOICES).toUpperCase();     //3
      while ( ! input.equals ("X"))                          //4
      {  if (input.equals ("A"))                             //5
         {  nextJob = new RepairOrder (IO.askLine (ENTER));  //6
            nextJob.setEstimatedTime (IO.askDouble (TIMEPROMPT));
            jobQueue.enqueue (nextJob);                      //8
            totalTime += nextJob.getEstimatedTime();         //9
            output.display ("\nHours remaining: " + totalTime);
         }                                                   //11
         else if (jobQueue.isEmpty())                        //12
            IO.say ("No jobs currently in the queue!");      //13
         else                                                //14
         {  nextJob = (RepairOrder) jobQueue.dequeue();      //15
            totalTime -= nextJob.getEstimatedTime();         //16
            output.display ("\nNext job: " + nextJob.toString()
                       + "\nHours remaining: " + totalTime); //18
         }                                                   //19
         input = IO.askLine (CHOICES).toUpperCase();         //20
      }                                                      //21
   }  //======================
}  //########################################################


class RepairShopApp
{
   public static void main (String[] args)
   {  new RepairShop().processRequests();                    //22
      System.exit (0);                                       //23
   }  //======================
}
```

6.9 More On Debugging Your Program: Tracing And Type-Checking

Sometimes your program produces the wrong answer, and sometimes it produces no answer at all (because it is stuck in an "infinite loop" or because it crashed). The problem is always either that (a) some variable has a different value from what you expected it to have, or (b) you do not really know what to expect from your own logic.

The latter can only be cured by diligent study and instruction. But in the former case, you can set up a manual trace of the state of every relevant variable. This is a table of the values those variables should have after each step of the program, for one particular set of input values.

Manual tracing

Figure 6.9 is a manual trace of the RepairShop program. You use a small amount of input, just enough to locate an error. For this sample run, assume you first enter a repair job for `sue` of 80 minutes, then a repair job for `bill` of 60 minutes, then process the next available repair job (which should be `sue`).

You put the name of each relevant variable or value at the top of the columns. You need not have a column for a variable that is only for output; instead you put lines in the table (boldfaced in Figure 6.9) showing what values are displayed at which points.

ACTION	TEST	nextJob	jobQueue	totalTime	input
jobQueue=			empty		
nextJob=		sue 1.33	<sue,1.33>		
totalTime=				1.33	
jobQueue=					
input=					A
test isn't "X"	true				
test is "A"	true				
nextJob=		bill 1.0			
jobQueue=			<sue,1.33>, <bill,1.0>		
totalTime=				2.33	
display				**2.33**	
input=					D
test isn't "X"	true				
test is "A"	false				
test jobQueue	false				
nextJob=		sue 1.33	<bill,1.0>		
totalTime=				1.0	
display				**1.0**	

Figure 6.9 Manual trace of RepairShop

Next you fill in the table one line at a time, applying the next step of the program and recording the new value of any variable that changed in that step, until you see that one variable has the wrong value. Figure 6.9 uses for the Queue's values <x,y> to denote a repair job for person `x` of `y` hours and puts the front of the Queue on the left. Note how the boldfaced lines show what you should see on the screen at that point, so you can easily check it against the actual run of the program.

Automated tracing

Sometimes this process does not bring out the error. The problem is that you are misreading some statement in the program or that you do not really understand what some statement does in this context. So your table's final answer is not the one the program produces. Once you find out just which one statement it is, you can usually see how you are misreading or misunderstanding it.

You may need an automated trace instead of a manual one. You should insert the following five statements in the program in Listing 6.10 right after the line whose number begins the `println` parameter. The first and fourth statements use the fact that string concatenation converts the object to `nextJob.toString()`:

```
   System.out.println (3 + " job=" + nextJob);
   System.out.println (4 + " time=" + totalTime);
   System.out.println (7 + " input=" + input);
   System.out.println (11 + " job=" + nextJob);
   System.out.println (24 + " input=" + input);
```

These five statements print out all the relevant values with a note of the point in the program where the message was written. They print to the terminal window so you do not have to keep clicking the OK button.

If you compare these values with your manual table, you should see the first point at which your table went wrong. Then you can see whether the input contained unexpected values or you misunderstood some statement (just before the debug message that disagreed with your table). Note that there is no need to print the values of the variables changed in lines 13, 19 and 20, because the very next statement displays them anyway.

Two difficulties with this technique are (a) the number of debugging message may be so large that they scroll off the screen before you can read them, and (b) you have to put `//` before each debugging line before you release the program for general use.

A solution is to have a Debug class with a class method you call to print the debugging output. Then the Debug class can (a) have a counter that does `showMessageDialog` after each fifteen lines of output, to give you a chance to read the messages, and (b) have a boolean variable you set `true` or `false`, with `true` required before any debugging messages appear. Then you simply set the boolean to `false` to disable all debugging messages before releasing the program for general use. The development of this class is an exercise.

Type-checking an expression

You should also know how to **type-check** the expressions in your program to make sure that they are acceptable to the compiler. This type-checking process consists of replacing each value or variable by its type and then applying the operators in the proper order to see what type each subexpression produces. The following two examples of type-checking boldface the subexpression to be evaluated next.

The `firstWord` method in the earlier Listing 6.4 has the phrase:

```
   if (itself.charAt (k) <= ' ')
```

1. Substitute for each variable and value in the expression its type to get:

   if (**String.charAt (int)** <= char)

2. Reduce the `charAt` call by first verifying that it is in fact a method in the String class and it really does have a single parameter of int type. You may then substitute for the call its return type, which is char:

```
    if (char <= char)
```

3. Reduce the inequality expression by first verifying that the values on both sides of the operator are the same type. You may then substitute for the expression its result type, which for `<=` is boolean. Since an if-statement requires a boolean value inside its parentheses, this is legal:

```
    if (boolean)
```

Another example of type-checking

Consider the following statement that assigns a value to a boolean variable. When doing type-checking, you start by putting in the default executor wherever possible:

```
    valueToReturn =  ! this.seesSlot()  ||  numLeft == 0;
```

1. Substitute for each variable and value in the expression its type to get:

```
    boolean =  ! Vic.seesSlot()  ||  int == int;
```

2. Reduce the equality expression by first verifying that the values on both sides of the operator are the same type. You may then substitute for the expression its result type, which is boolean:

```
    boolean =  ! Vic.seesSlot()  ||  boolean;
```

3. Reduce the `seesSlot` method call by first verifying that it is in fact a method in the Vic class with no parameters. You may then substitute for the call its return type, which is boolean:

```
    boolean =  ! boolean  ||  boolean;
```

4. Reduce the not-expression by first verifying that `!` is applied to a boolean value as required. You may then substitute for the expression its result type, which for the not-operator is boolean:

```
    boolean =  boolean  ||  boolean;
```

5. Reduce the or-expression by first verifying that `||` is applied to two boolean values as required. You may then substitute for the expression its result type, which for the or-operator is boolean:

```
    boolean =  boolean;
```

Since this is an assignment to a variable of the same type, it is legal.

If none of this works, it is time to call your instructor at his home and ask for help. Hopefully this is not on a Sunday evening when the assignment is due Monday and you put off doing it until the last day.

Exercise 6.54* Write the Debug class described in this section and describe the changes in Listing 6.9 that allow you to use the Debug class properly.
Exercise 6.55* Apply the type-checking process to line 7 of Listing 6.10.
Exercise 6.56* Apply the type-checking process to the first statement of Listing 6.7.
Exercise 6.57** Apply the type-checking process to line 7 of Listing 6.5.

*6.10 More On Random, NumberFormat, and DecimalFormat (*Sun Library)*

This section describes methods from the Random, NumberFormat, and DecimalFormat standard library classes that can be quite useful, though they are not mentioned elsewhere in this book. Since this book gives only a selection from the thousands of methods in the hundreds of classes of the standard library, you should research additional features of Java not covered in this book at `http://java.sun.com/docs`. The most important is the **API** (Application Programming Interface) link on that page.

Random objects (from the java.util package)

If you create a Random object using `randy = new Random(someLongValue)`, you set a 32-bit long "seed" value to be used in calculating pseudorandom numbers. It gives the same sequence of "random" numbers on each run of the program, which is nice for debugging. If you use `new Random()`, it sets the seed from the current time using `System.currentTimeMillis()`. You can also change the seed used to produce a sequence of random numbers with the statement `randy.setSeed(someLongValue)`.

The basic method used internally by the Random class is `next(anInt)`, which calculates a new seed from the old one and then returns the number formed by the first `anInt` bits of the seed -- a "randomly chosen number". You may use the following instance methods to obtain pseudorandom numbers, all uniformly distributed except for `nextGaussian`, where `randy` could be any Random object:

- `randy.nextInt()` returns a 32-bit number `next(32)`, since int values have 32 bits. It will be a negative number about half the time.
- `randy.nextInt(limitInt)` returns essentially `next(31) % limitInt`, with minor adjustments. It throws an IllegalArgumentException if n is not positive.
- `randy.nextLong()` returns a 64-bit number `next(32) *` 2^{32} `+ next(32)`.
- `randy.nextDouble()` returns a number from 0.0 to 0.9999..., namely `next(26) /` 2^{26} `+ next(27) /` 2^{53} since the digit part of a double value has 53 bits (the other 11 bits are used to keep track of the exponent part).
- `randy.nextFloat()` returns `next(24) /` 2^{24}, since a float value is a decimal number stored with 32 bits and the digit part of that value has 24 bits.
- `randy.nextGaussian()` returns a double value that has a standard normal distribution with mean of 0.0 and standard deviation of 1.0, for use in statistics.

For instance, the following prints 100 double values in the range -10 to 10:

```
for (int k = 0;  k < 100;  k++)
   System.out.println (randy.nextDouble() * 20 - 10);
```

NumberFormat objects (from the java.text package)

You sometimes want to print a number rounded to a particular number of decimal places. For instance, if you were to print 0.1 + 0.2, you would not see 0.3; you would see 0.30000000000000004. This is because numbers are stored in binary form. Therefore, Java variables do not store fractional numbers exactly unless the only division they involve is by a power of two. 0.2 is 1/5, and 5 is not a power of two.

You could use a standard library class named **NumberFormat** from the `java.text` package. You would first have to create a NumberFormat object as follows:

```
NumberFormat form = NumberFormat.getNumberInstance();
```

Then you need to set the maximum and minimum number of digits after the decimal point: `setMaximumFractionDigits` causes the NumberFormat object to round off if the String representation of the number has too many digits after the decimal point; and `setMinimumFractionDigits` causes it to add extra zeros if the String representation of the number does not have enough digits after the decimal point. The actual formatting is done by calling the `format` instance method, which has a double parameter and returns a String value. For instance, to have the int or double value in `x` written with exactly two digits after the decimal point, you could use these three statements:

```
form.setMaximumFractionDigits (2);
form.setMinimumFractionDigits (2);
System.out.println (form.format (x));
```

You could modify Listing 6.6 to print the reciprocals of the integers 1 to 100, correct to four decimal places. Put the following among the first few statements of Listing 6.6:

```
NumberFormat form = NumberFormat.getNumberInstance();
form.setMaximumFractionDigits (4);
form.setMinimumFractionDigits (2);
```

If you then replace the for-statement by the following, you will see 1.00 as the reciprocal of 1, 0.50 as the reciprocal of 2, 0.25 for 4, 0.125 for 8, and all other outputs will be rounded to exactly four digits after the decimal point:

```
for (int k = 1;  k <= 100;  k++)
   table += "\n" + k + "\t" + form.format (1.0 / k);
```

You may also have the part of the number before the decimal point be separated by commas into groups of three digits by calling the instance method `setGroupingUsed` with a boolean parameter: `true` adds commas, `false` omits them:

```
form.setGroupingUsed (true);
System.out.println (form.format(4175238)); // prints 4,175,238
```

DecimalFormat objects (from the java.text package)

DecimalFormat is a subclass of NumberFormat. You may use it to easily specify the number of digits in the string representation of a number. You could create three DecimalFormat objects as follows:

```
DecimalFormat withCommas = new DecimalFormat ("#,###");
DecimalFormat withThree = new DecimalFormat ("00.000");
DecimalFormat dollars = new DecimalFormat ("$0.00");
```

- `withCommas.format(x)` for a given double, long, or int value `x` returns a String value that is rounded to the nearest integer and has a comma every three digits.
- `withThree.format(x)` returns a String value that has at least two digits before the decimal point and exactly three digits after the decimal point (i.e., rounded if it naturally has more than three digits, with trailing zeros as needed to pad it out to three digits). For instance, it returns `"04.700"` when `x` is 4.7 and returns `"-231.625"` when `x` is -231.62547.
- `dollars.format(x)` returns a String value that has exactly two digits after the decimal point and a "floating dollar sign" immediately before the first digit of the number; for instance, it returns `"$14.42"` when `x` is 14.423.

6.11 Review Of Chapter Six

Listing 6.1, Listing 6.3, and Listing 6.4 illustrate most of the Java language features introduced in this Chapter.

About the Java language:

- Variables of **double** type store numbers with decimal points in scientific notation with about 15-decimal-digit accuracy, using 8 bytes (1 **byte** can store 8 binary digits).
- In the sequence of primitive types char => int => long => double from "narrow" to "wide", you may assign a narrower value to a wider variable, which **promotes** the narrower value to a wider one. But assigning a wider value to a narrower variable (such as a double value to an int variable) requires that you cast it, e.g., put an `(int)` **cast** in front of a double value to convert it to an int value. If you combine two different kinds of these values with an operator, the narrower one is promoted. The promotion of a char value is to the int value corresponding to its **Unicode** value (0 to 65,535).
- `x += y` means `x = x + y`, and similarly for `-=`, `*=`, `/=`, and `%=`.
- An expression of the form `testing ? oneThing : anotherThing` uses the **conditional operator** to obtain a value. Both the question mark and the colon are required. If `testing` is true then `oneThing` gives the value; otherwise `anotherThing` gives the value. Those two parts on either side of the colon have to be of the same type (int or boolean or Person or whatever).
- The **members of a class** X are the variables, methods, and classes declared inside of X (except of course those declared inside of any method or class that is inside X). A class declared inside another class with the heading `private static class Whatever` is a **nested** class of the containing class. This affects nothing but the visibility of various class members, so it is primarily done to maintain encapsulation.
- A method **throws an Exception** when it notifies the runtime system of a problem that crashes the program (unless you have special statements that are discussed in Chapter Nine). Examples are an **ArithmeticException** (caused by trying to divide an integer by zero), a **NumberFormatException** (thrown by an attempt to parse a string of characters as a numeral when it is not), an **IndexOutOfBoundsException** (thrown by e.g. `s.charAt(k)` if it is false that `0 <= k < s.length()`), a **ClassCastException** (thrown by `(C)x` when `x` is not in class C or in a subclass of C), and a **NullPointerException** (thrown by e.g. `x.getName()` when `x` is null).
- **Comparable** is a category of classes, called an **interface**. It specifies you may have `implements Comparable` in your class heading if your class or its superclass has a method with the heading `public int compareTo(Object ob)`. You may then assign objects of your class to a Comparable variable.
- If Sub is a subclass of Super, you may assign a Sub value to a Super variable. At runtime, a call of a method in the Super class with that Sub value as the executor will actually call the method in the Sub class that overrides it (if any).
- To tell the compiler that an object variable or value will contain at runtime a reference to an object from a subclass Sub of the variable's declared class, put a `(Sub)` cast in front of the variable or value. This **class cast** operation can also be used to tell the compiler that the object is of a class Sub that implements a specific interface.
- Variables of **char** type store individual characters. Character literals have apostrophes, as in `'x'`. Five special ones are the newline `'\n'`, tab `'\t'`, quote `'\"'`, backspace `'\b'`, and backslash `'\\'` characters.
- The Unicode encoding assigns consecutive integers to `'A'` through `'Z'`, also to `'a'` through `'z'`, and also to `'0'` through `'9'`. Characters with Unicode values less than or equal to that of a blank are called **whitespace**.

- A **long** value is a whole-number value stored using 64 bits (8 bytes), so it is roughly plus-or-minus 8 billion billion. You may assign a long value to a double variable without an explicit cast, but not to an int or char variable. The boolean, char, int, long, and double categories of values are called **primitive types**.

Other vocabulary to remember:

- We discussed two common loop patterns: A **sentinel-controlled loop** is a loop that repeats for each value in a sequence of values until you reach a special value, different from normal values, signaling the end of the sequence. A **count-controlled loop** is a loop that repeats until a counter reaches a specified value.
- Some common actions to take inside a loop, illustrated in this chapter, are the Count-cases looping action, the All-A-are-B looping action, and the Some-A-are-B looping action. The latter two perform a **sequential search** for a value that does or does not have a particular property.
- A **String literal** is a sequence of characters enclosed in quotes, e.g., `"5 \t x"`.
- Assigning a char value to an int variable or an int value to a double variable causes a **widening conversion**; the compiler does this automatically. The opposite is a **narrowing conversion**; it requires a cast.
- A class of objects is formed by **composition** if its only or its primary instance variable is an object of another class. Some authors define "composition" to mean that at least one instance variable is an object, which includes all classes where a String is an instance variable.
- A **crash-guard** is a condition that is tested to avoid evaluating an expression that would crash the program if the crash-guard were not tested first.
- A **queue** is a list of things for which the primary operations are adding a new value and removing an old value. The latter operation always removes the value that has been on the list for the longest period of time.
- The **object design** for a piece of software is a list of the major object classes it uses and the services that each such class offers. These services can be described by giving the headings of the public methods together with comments saying what each does. Developing the main logic of the software often helps you decide what objects you need.
- The **Model/View/Controller pattern** is explained in Section 6.8.
- A **trace** of a program is a listing of the values that the most important variables have at various points in the execution of the program. It is used for studying and debugging the program.

About the java.lang.String class:

- `someString.compareTo(aString)` is a String query that returns an int value. If `someString` comes after/before `aString` in **lexicographical order** (based on the Unicode values of their characters), then `someString.compareTo(aString)` will be some positive/negative integer, respectively.
- `someString.charAt(indexInt)` is a String query that returns the character at position `indexInt` (numbered from 0 up). `indexInt` is an int value for which `0 <= indexInt < someString.length()`.
- `someString.substring(startInt)` produces the substring of characters starting at index `startInt` in `someString` and going to the end. `startInt` is an int value for which `0 <= startInt <= someString.length()`.

About the java.lang.Math class:

- `Math.PI` is the ratio of the circumference of a circle to its diameter.
- `Math.E` is the natural base of logarithms, roughly 2.718281828.
- `Math.sqrt(x)` returns the square root of the double value `x`.
- `Math.abs(x)` returns the absolute value of `x`. It returns an int when `x` is an int value, a double when `x` is double, a long when `x` is long.
- `Math.min(x, y)` is the smaller of `x` and `y`. It returns an int value when both parameters are ints, a double when both are double, a long when both are long.
- `Math.max(x, y)` is the larger of `x` and `y`. It returns an int value when both parameters are ints, a double when both are double, a long when both are long.
- `Math.pow(x, y)` returns x-to-the-power-y, where `x` and `y` are doubles.
- `Math.random()` returns a "random" double number `r` such that `0 <= r < 1`.
- `Math.cos(x)` returns the cosine of the double radian value `x`.
- `Math.sin(x)` returns the sine of the double radian value `x`.
- `Math.log(x)` is the natural logarithm of the double value `x`, that is, to base e.
- `Math.exp(x)` is e to the power `x`, so `Math.exp(Math.log(x))` is `x`.

About other Sun standard library classes:

- `Double.parseDouble(someString)` returns the double equivalent of the numeral (which can be in ordinary integer or decimal form, or in **scientific notation** such as 6.3E+4). It throws a NumberFormatException if the numeral is ill-formed.
- `System.currentTimeMillis()` returns the long current time in milliseconds.
- `new JTextArea(rowsInt, colsInt)` creates a new JTextArea object, a rectangular output region with a specified number of rows and columns of characters.
- `someJTextArea.setText(someString)` makes the information that appears in a JTextArea executor be the specified String value.
- `someJTextArea.append(someString)` adds text to that information.
- `new JScrollPane(someJTextArea)` creates a new JScrollPane object wrapped around a given JTextArea. A JScrollPane or a JTextArea can be the second parameter of a `JOptionPane.showMessageDialog` method call.

Answers to Selected Exercises

6.1
```
class FindAverage
{    public static void main (String[ ] args)
     {    JOptionPane.showMessageDialog (null, "The average of three numbers");
          double first = Double.parseDouble (JOptionPane.showInputDialog ("Enter #1:"));
          double second = Double.parseDouble (JOptionPane.showInputDialog ("Enter #2:"));
          double third = Double.parseDouble (JOptionPane.showInputDialog ("Enter #3:"));
          JOptionPane.showMessageDialog (null, "Their average is "
                                   + ((first + second + third) / 3));
          System.exit (0);
     }
}
```
6.2
```
class Gasoline
{    public static void main (String[ ] args)
     {    JOptionPane.showMessageDialog (null, "Convert marks/liters to dollars/gallon");
          String s = JOptionPane.showInputDialog ("Enter price of one DeutschMark:");
          double marks = Double.parseDouble (s);
          s = JOptionPane.showInputDialog ("Enter price of one liter of gas in DM:");
          double gasPrice = Double.parseDouble (s);
          double cost = marks * gasPrice / 0.264;
          JOptionPane.showMessageDialog (null, "The dollar cost per gallon is " + cost);
          System.exit (0);
     }
}
```
6.3 total = total + turn can be written as total += turn.
power = 2 * power can be written as power *= 2.
given = given / 2 can be written as given /= 2.

6.4
```
class ConvertDays
{    public static void main (String [] args)
     {    String input = JOptionPane.showInputDialog ("Enter a number of days");
          double days = Double.parseDouble (input);
          double hours = 24.0 * days;
          double minutes = 60.0 * hours;
          double seconds = 60.0 * minutes;
          JOptionPane.showMessageDialog (null, days + " days equals " + hours
                 + " hours, or " + minutes + " minutes, or " + seconds + " seconds.");
          System.exit (0);
     }
}
```
6.9
```
public static int askNonNeg (String prompt)
{    int valueToReturn = askInt (prompt);
     while (valueToReturn < 0)
          valueToReturn = askInt (prompt);
     return valueToReturn;
}
```
6.10 In Listing 4.5: `return (itsHour < 10) ? "0" + itsHour + itsMin : "" + itsHour + itsMin;`
In Listing 4.6: `itsUsersNumber = (s == null || s.equals ("")) ? -1 : Integer.parseInt (s);`

6.14 Replace "return" by "IO.say" in three places and put parentheses around the phrase that follows "return"; then replace "String" by "void" in the heading.

6.15 JOptionPane.showMessageDialog (null, "" + x), because showMessageDialog(null, x) will not compile, since x is an int value, not a String value.

6.16 IO.say (s.substring (3,6)), since the fourth character is numbered 3 and the seventh is numbered 6.

6.17
```
public static boolean sameFirst5 (String one, String two)
{    return one != null && two != null && one.length() >= 5 && two.length() >= 5
                    && one.substring (0, 5).equals (two.substring (0, 5));
}
```
6.18
```
public static int indexOf (String big, String little)
{    int len = little.length();   // to speed up execution
     int lastToLookAt = big.length() - len;  // also to speed up execution
     for (int k = 0;  k <= lastToLookAt;  k++)
     {    if (big.substring (k, k + len).equals (little))
               return k;
     }
     return -1;
}
```
6.22 s.charAt(2) is 'g', s.substring(2) is "gorithm", and s.substring(2,6) is "gori".

6.23 itself.charAt (0) is whitespace, so the value returned is itself.substring (0,0), which is "".

6.24
```
public int countRange (char lo, char hi)
{    int count = 0;
     for (int k = 0;  k < itself.length();  k++)
     {    if (itself.charAt (k) >= lo && itself.charAt (k) <= hi)
               count++;
     }
     return count;
}
```
6.25
```
public char biggestChar()
{    char valueToReturn = (char) 0;
     for (int k = 0;  k < itself.length();  k++)
     {    if (itself.charAt (k) > valueToReturn)
               valueToReturn = itself.charAt (k);
     }
     return valueToReturn;
}
```
6.26
```
public void trimRear()
{    int k = itself.length() - 1;
     while (k >= 0 && itself.charAt (k) <= ' ')
          k--;
     itself = itself.substring (0, k + 1);  // up to and including k; works even if k == -1
}
```

6.27
```
public boolean hasWhite()
{       for (int k = 0;  k < itself.length();  k++)
        {       if (itself.charAt (k) <= ' ')
                        return true;
        }
        return false;  // since it contains no whitespace
}
```
6.28
```
public int indexOf (char par)
{       for (int k = 0;  k < itself.length();  k++)
        {       if (itself.charAt (k) == par)
                        return k;
        }
        return -1;  // since par is not in the string
}
```
6.29
```
public String initDigits()
{       for (int k = 0;  k < itself.length();  k++)
        {       if (itself.charAt (k) < '0' || itself.charAt (k) > '9')
                        return itself.substring (0, k);  // not including itself.charAt (k)
        }
        return itself;  // since the string has nothing but digits
}
```
6.36
```
public static char toUpperCase (char par)
{       return (par < 'a' || par > 'z')  ?  par  :  (char) (par - 'a' + 'A');
}
```
6.37
```
public static double largestOf4 (double x, double y, double z, double w)
{       return Math.max (Math.max (x, y), Math.max (z, w));
}
```
6.38
```
class PrintRoots
{       public static void main (String[ ] args)
        {       double input = IO.askDouble ("Enter a number to find many roots of");
                double fourth = Math.sqrt (Math.sqrt (input));
                IO.say ("square: " + Math.sqrt (input) + ", fourth: " + fourth
                                + ", eighth: " + Math.sqrt (fourth));
                System.exit (0);
        }
}
```
6.39 In the seesTwoEmpty method, the statement if (seesCD()) return false; is guarded by the initial if (! seesSlot()), a Type 3 crash guard. The test if (seesSlot()) is a Type 1 crash guard for the statement beginning if (! seesCD()). hasTwoOnStack has no guards.

6.45
```
public void setEstimatedTime()
{       if (itsEstimatedTime == 0)
                itsEstimatedTime = IO.askDouble ("Enter a new estimated time:");
}
```
6.46 Add these two statements to the constructor: itsID = theID; theID = (char) (theID + 1);
Also add the following declarations to the RepairOrder class:
```
private static char theID = 'A';
private char itsID;
public static char getID()
{       return itsID;
}
```
6.50
```
public RepairOrder (String description, double time)
{       if (description != null)
                getTheInfo (description);  // method whose body is the body of the if-statement in the
        itsEstimatedTime = time;       // other constructor.  Have the other constructor call getTheInfo too.
}
```
Replace the two lines after the first if condition by the following:
```
nextJob = new RepairOrder (IO.askLine (ENTER), IO.askDouble (TIMEPROMPT));
```
Replace the declaration of nextJob by the following:
```
private RepairOrder nextJob = new RepairOrder (IO.askLine (ENTER),
                                        IO.askDouble (TIMEPROMPT));
```
replace the first statement of the RepairShop constructor by the following:
```
totalTime = nextJob.getEstimatedTime();
```
Also, omit the initialization of totalTime.