Chapter 7

Repetition Statements

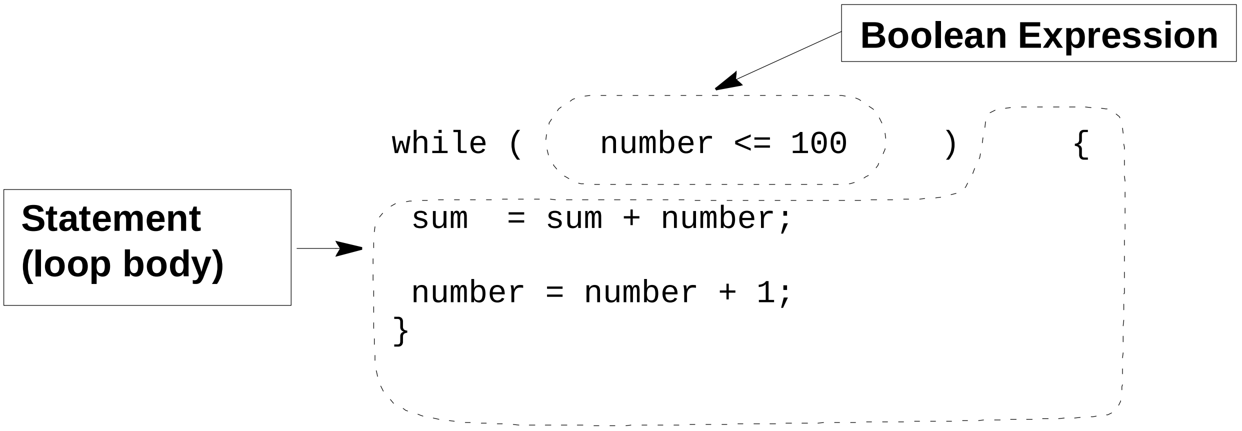
OBJECTIVES

After you have read and studied this chapter, you should be able to

- Implement repetition control in a program using while statements.
- Implement repetition control in a program using do–while statements.
- Implement repetition control in a program using for statements.
- Nest a loop repetition statement inside another repetition statement.
- Choose the appropriate repetition control statement for a given task.
- Prompt the user for a yes–no reply using the ResponseBox class from the javabook package.
- Output formatted data using the Format class from the javabook package.
- (Optional) Write simple recursive methods

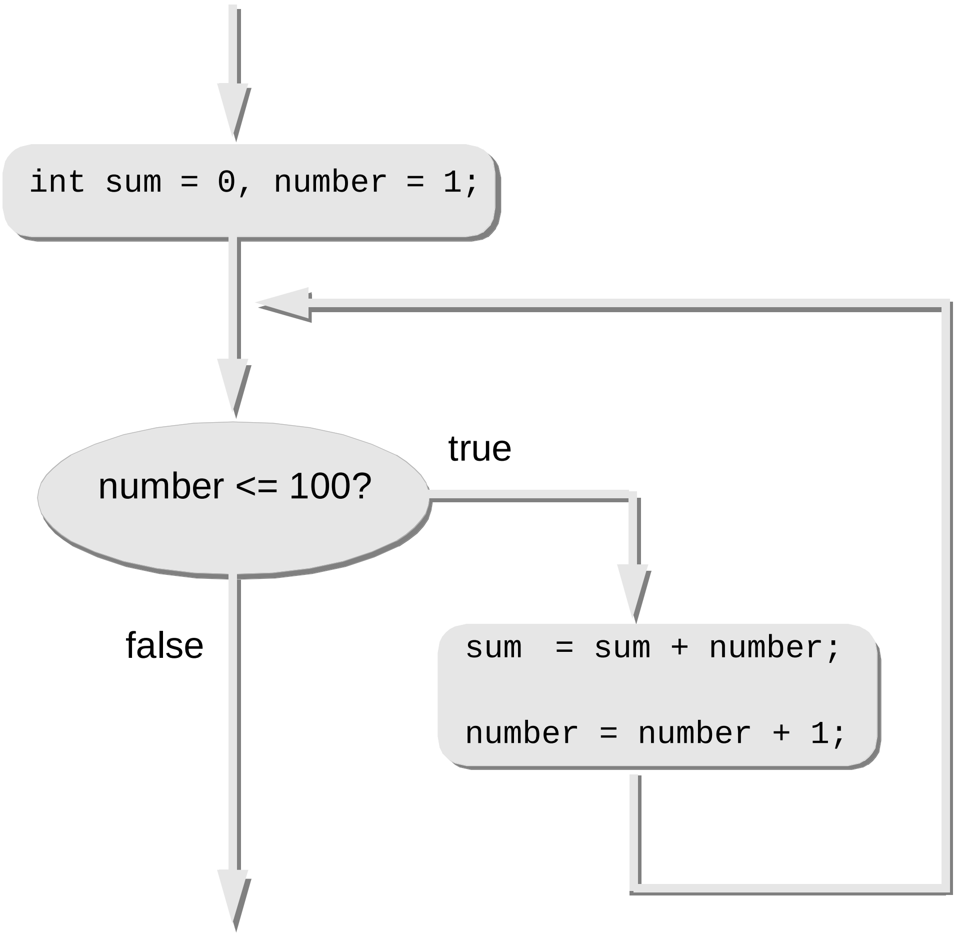
McGraw-Hill Publishing

FIGURE 7.1 Correspondence of the example **while** statement to the general format.



```
while (  number <= 100  )   {
  sum  = sum + number;

  number = number + 1;
}
```

FIGURE 7.2 A diagram showing the control flow of a **while** statement.



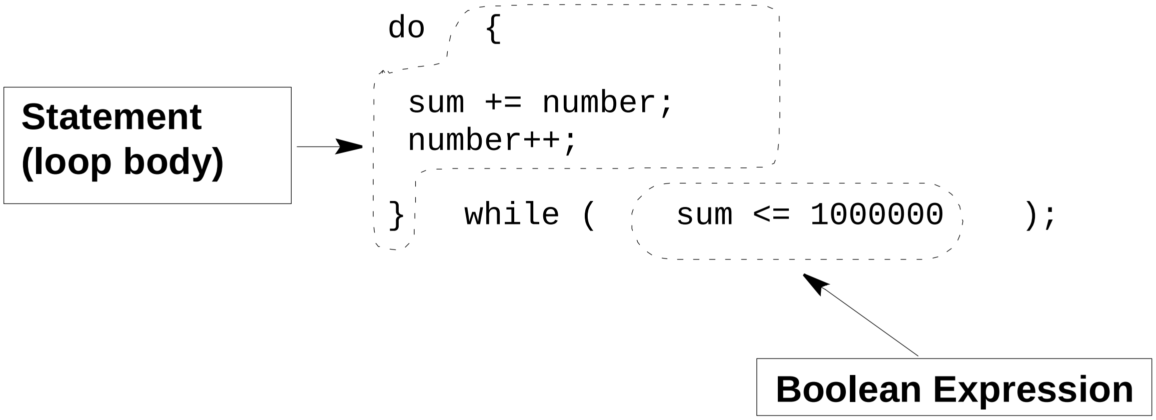
```
int sum = 0, number = 1;
```

```
sum  = sum + number;

number = number + 1;
```

TABLE 7.1 Shorthand assignment operators.

Operator	Usage	Meaning
`+=`	`a += b;`	`a = a + b;`
`-=`	`a -= b;`	`a = a - b;`
`*=`	`a *= b;`	`a = a * b;`
`/=`	`a /= b;`	`a = a / b;`
`%=`	`a %= b;`	`a = a % b;`

FIGURE 7.3 Correspondence of the example **do–while** statement to the general format.



```
do  {

  sum += number;
  number++;

}  while (  sum <= 1000000   );
```

FIGURE 7.4 A diagram showing the control flow of the **do–while** statement.

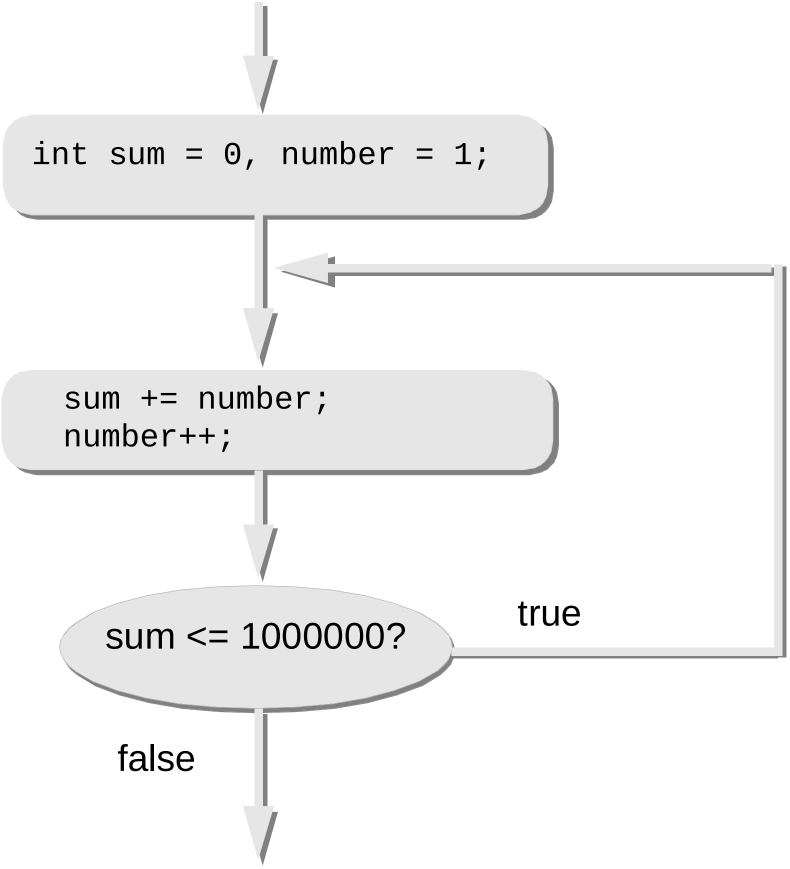
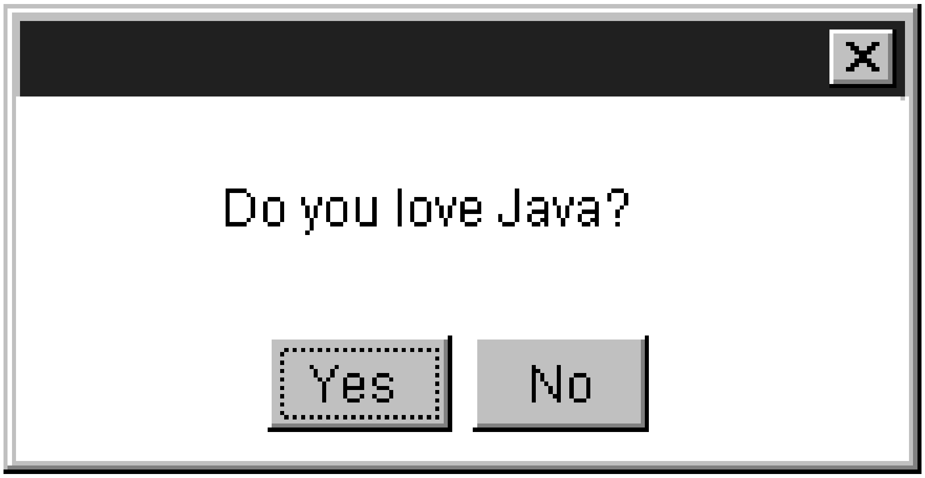
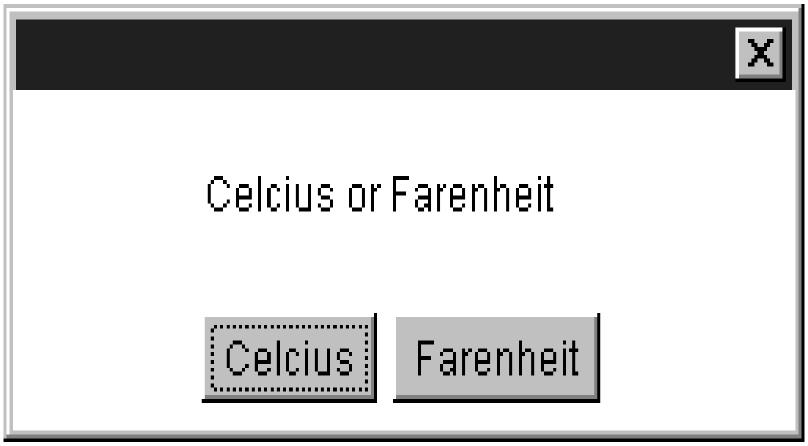


FIGURE 7.5 A **ResponseBox** dialog box with the prompt “Do you love Java?”



Do you love Java?

FIGURE 7.6 The **ResponseBox** object with user-specified button labels.



Celcius or Farenheit

TABLE 7.2 A list of **ResponseBox** methods.

CLASS:	**ResponseBox**	
Method	**Argument**	**Description**
`<constructor>`	`MainWindow`	Creates a ResponseBox object.

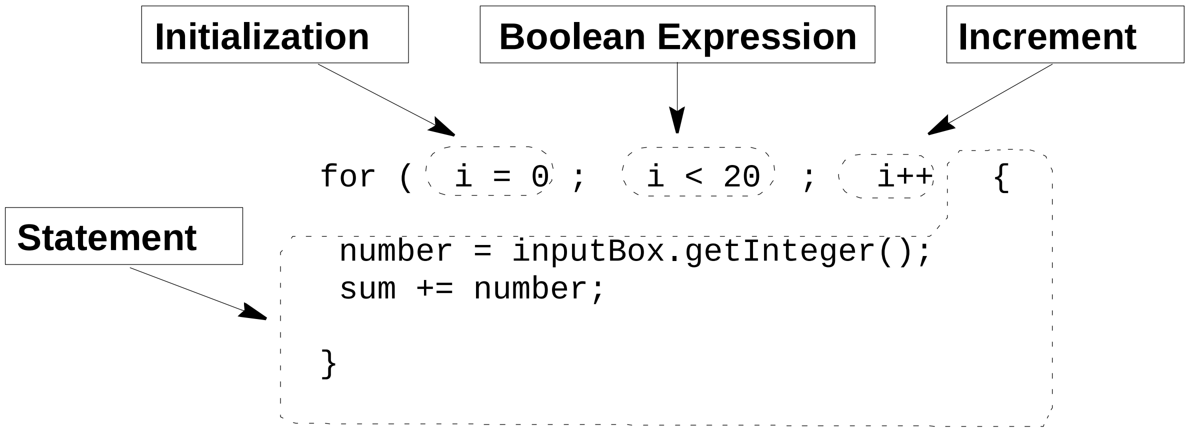
`<constructor>`	`MainWindow,`
	`int`
		Creates a ResponseBox object with N (the second argument) buttons, 1 <= N <= 3. If an invalid N is passed, then the object will include one button.

`prompt`	`String`	Prompts the user with the text passed as an argument. Returns an integer that identifies the clicked button. See the explanation of the class constants.

`setLabel`	`int,`
	`String`
		Sets the label of the designated button with the passed String. The first argument identifies the button. See the explanation of the class constants.

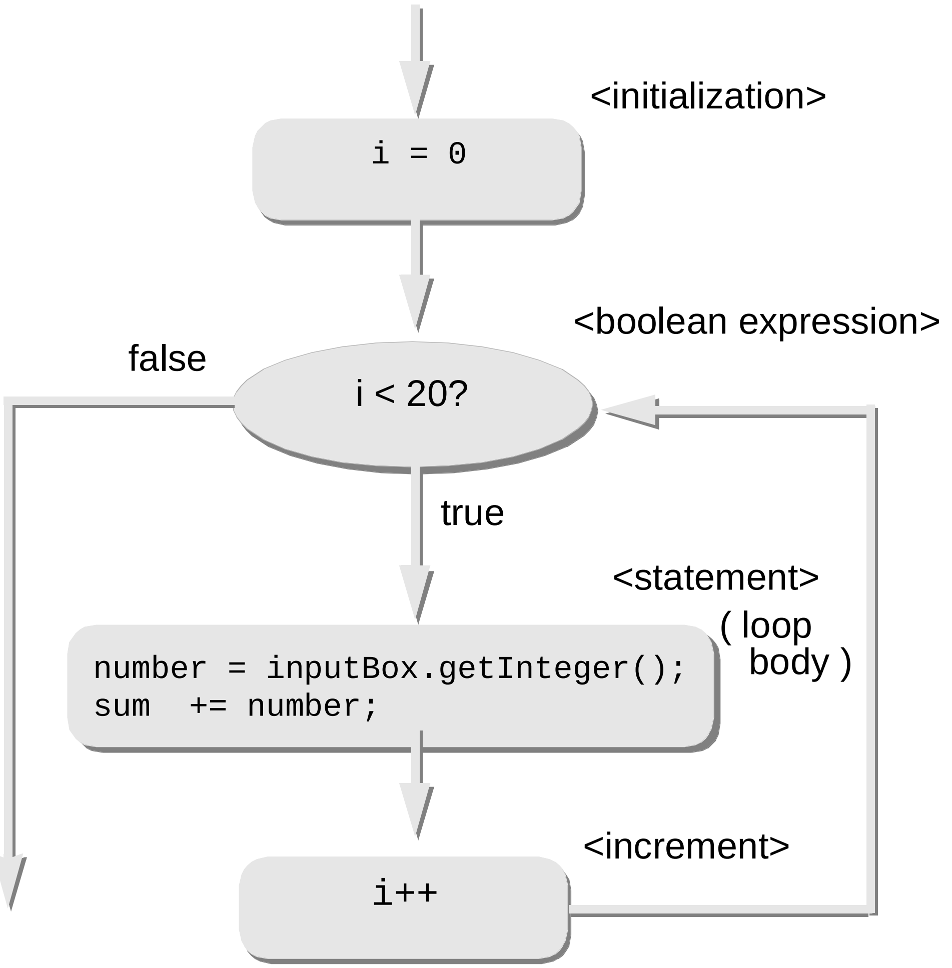
Class Constant		**Description**
`YES`		This value identifies the Yes button.
`NO`		This value identifies the No button.
`BUTTON1`		This value identifies the leftmost button. The value of BUTTON1 is equal to the value of YES.
`BUTTON2`		This value identifies the middle button. Note: the middle button becomes the rightmost button if there are only two buttons. The value of BUTTON2 is equal to the value of NO.
`BUTTON3`		This value identifies the rightmost button when the ResponseBox includes three buttons.

FIGURE 7.7 Correspondence of the example **for** statement to the general format.



```
for (  i = 0 ;  i < 20  ;  i++   {

  number = inputBox.getInteger();
  sum += number;

}
```

FIGURE 7.8 A diagram showing the control flow of the example for statement.



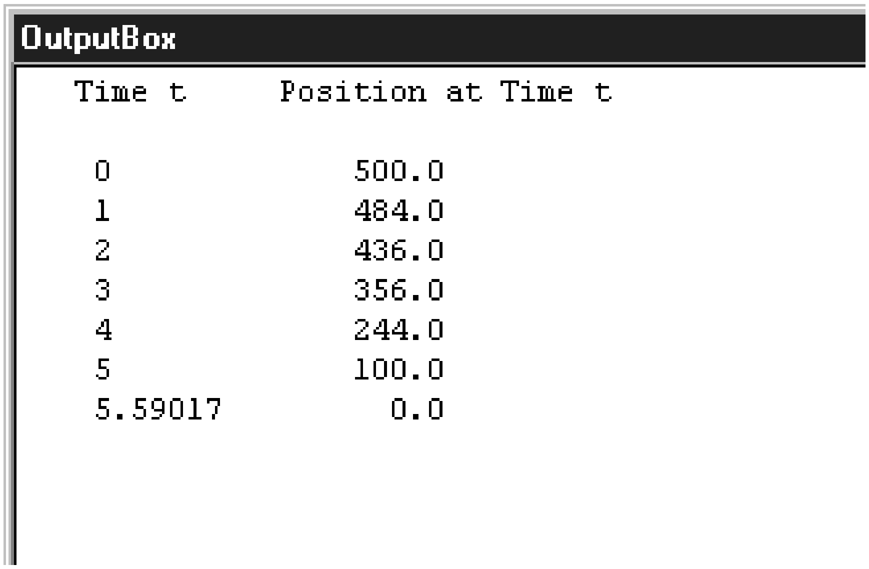
```
i = 0
```

```
i < 20?
```

```
number = inputBox.getInteger();
sum  += number;
```

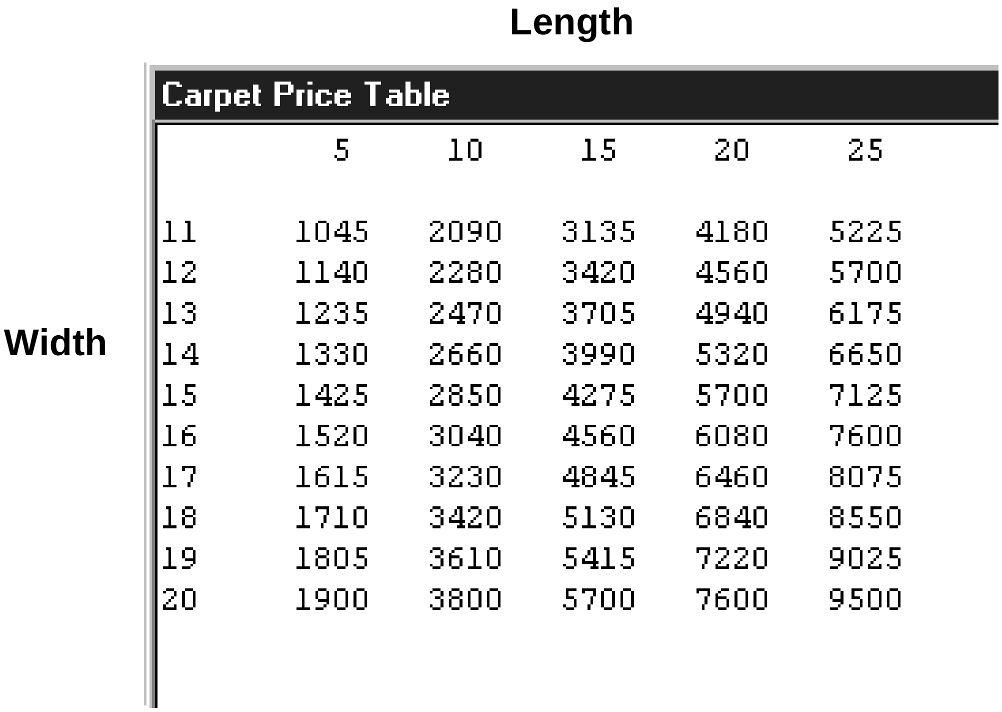
```
i++
```

FIGURE 7.9 The positions of a watermelon dropped from a height of 500 feet.



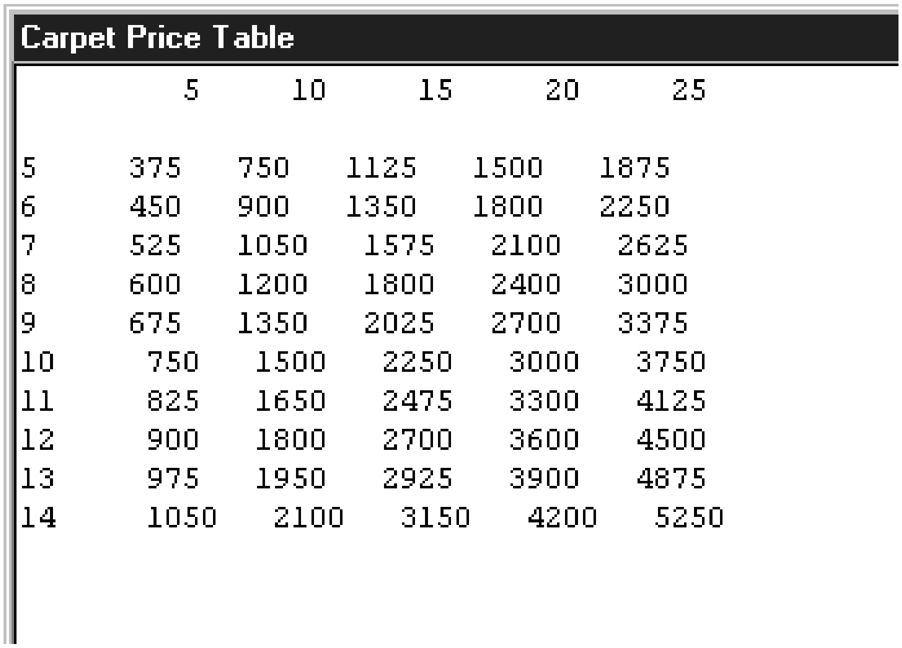
```
   Time t     Position at Time t

    0              500.0
    1              484.0
    2              436.0
    3              356.0
    4              244.0
    5              100.0
    5.59017          0.0
```

FIGURE 7.10 The price table for carpets ranging in size from 11 × 5 feet to 20 × 25 feet whose unit price is $19 per square foot.



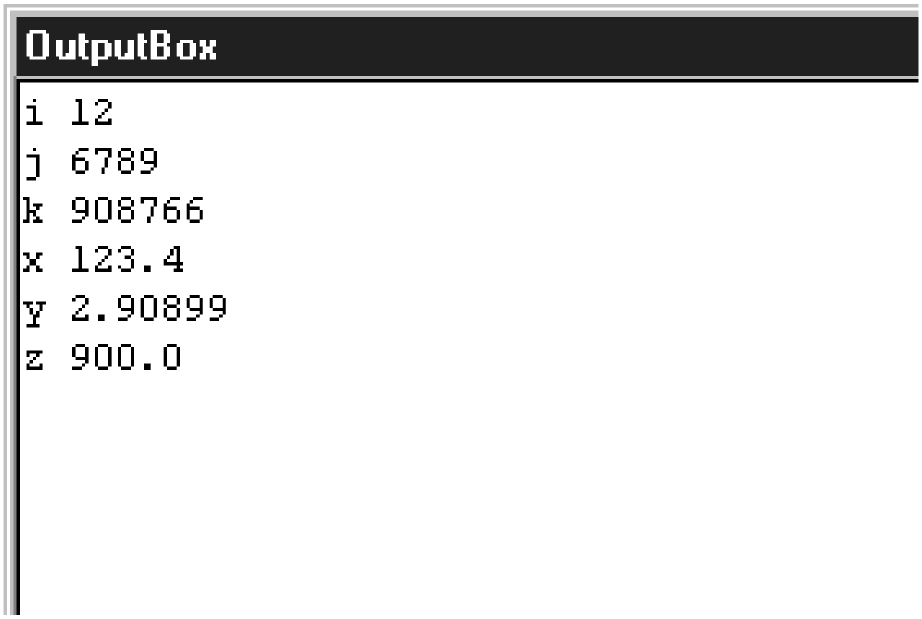
```
             5      10      15      20      25


11       1045    2090    3135    4180    5225
12       1140    2280    3420    4560    5700
13       1235    2470    3705    4940    6175
14       1330    2660    3990    5320    6650
15       1425    2850    4275    5700    7125
16       1520    3040    4560    6080    7600
17       1615    3230    4845    6460    8075
18       1710    3420    5130    6840    8550
19       1805    3610    5415    7220    9025
20       1900    3800    5700    7600    9500
```

FIGURE 7.11 The price table for carpets with $15 per square foot and width ranging from 5 through 14.



```
          5     10     15     20     25

5     375   750   1125   1500   1875
6     450   900   1350   1800   2250
7     525   1050   1575   2100   2625
8     600   1200   1800   2400   3000
9     675   1350   2025   2700   3375
10     750   1500   2250   3000   3750
11     825   1650   2475   3300   4125
12     900   1800   2700   3600   4500
13     975   1950   2925   3900   4875
14     1050   2100   3150   4200   5250
```

FIGURE 7.12 Unformatted output of integers and floats.



```
i 12
j 6789
k 908766
x 123.4
y 2.90899
z 900.0
```

TABLE 7.3 A list of **Format** methods.

CLASS:	**Format**	
Class Method	**Argument**	**Description**

`leftAlign`	`int,`
	`long` or `int` or `String`
		The first argument designates the field width. The second argument is left aligned in the given field. The method return the formatted value as a String.

`leftAlign`	`int,`
	`int,`
	`double` or `float`
		The first argument designates the field width. The second argument designates the decimal places. The third argument is left aligned in the given field. The method return the formatted value as a String.

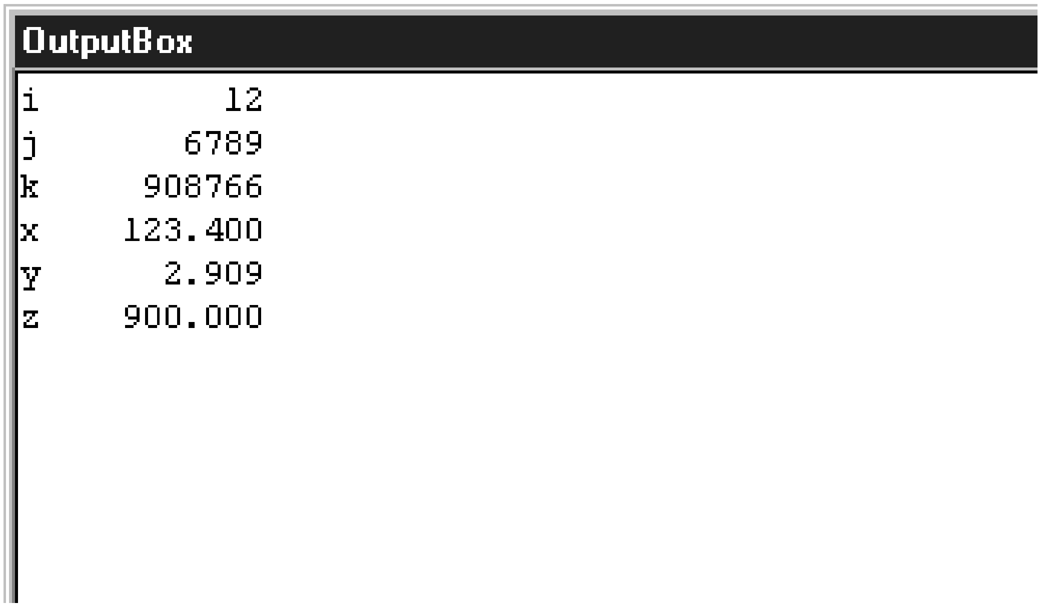
`centerAlign`	`int,`
	`long` or `int` or `String`
		Same as the first version of leftAlign, but with the center alignment.

`centerAlign`	`int,`
	`int,`
	`double` or `float`
		Same as the second version of leftAlign, but with the center alignment.

`rightAlign`	`int,`
	`long` or `int` or `String`
		Same as the first version of leftAlign, but with the right alignment.

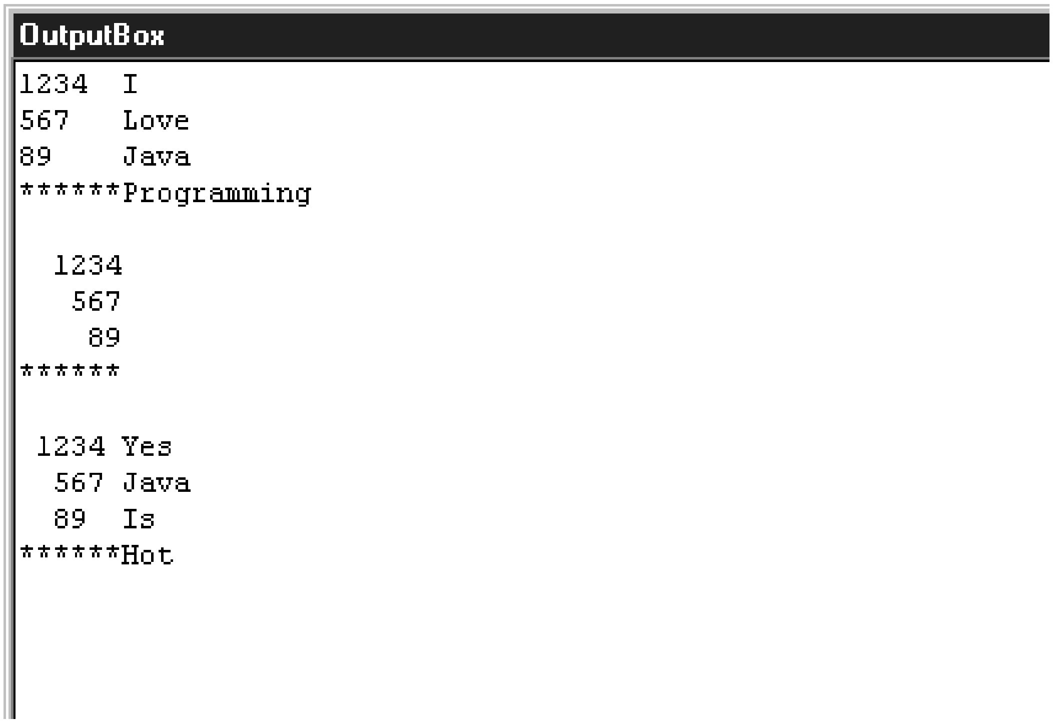
`rightAlign`	`int,`
	`int,`
	`double` or `float`
		Same as the second version of leftAlign, but with the right alignment.

FIGURE 7.13 Formatted output of integers and floats.



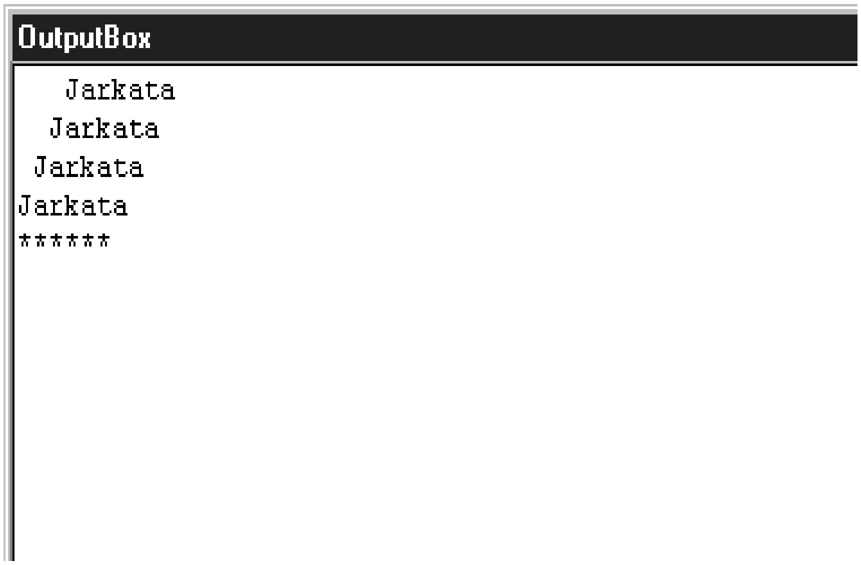
```
i          12
j        6789
k      908766
x     123.400
y       2.909
z     900.000
```

FIGURE 7.14 Formatted output of integers, demonstrating various alignments.



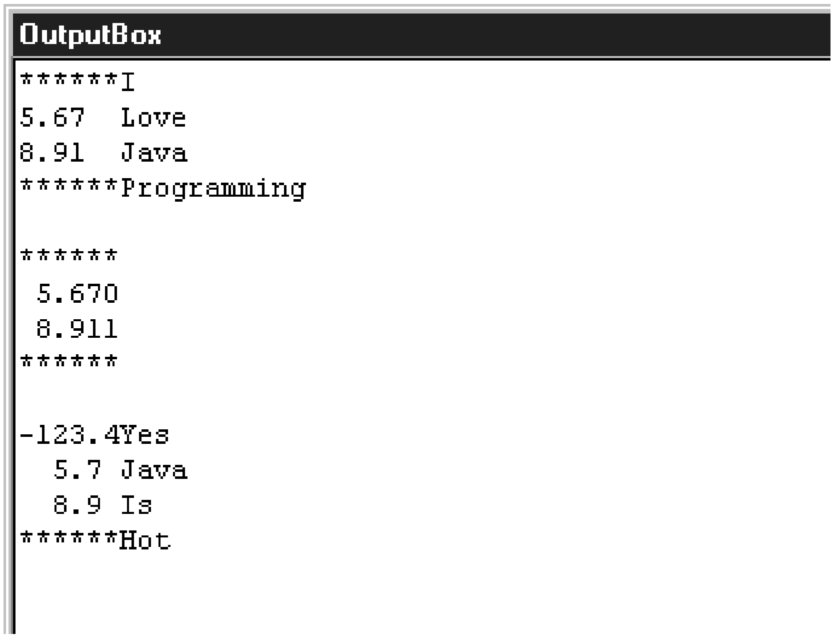
```
1234  I
567   Love
89    Java
******Programming

  1234
   567
    89
******

 1234 Yes
  567 Java
  89  Is
******Hot
```

FIGURE 7.15 Formatted output of the string "Jakarta".



```
   Jarkata
  Jarkata
 Jarkata
Jarkata
******
```

FIGURE 7.16 Formatted output of real numbers, demonstrating various alignments.



```
******I
5.67  Love
8.91  Java
******Programming

******
 5.670
 8.911
******

-123.4Yes
  5.7 Java
  8.9 Is
******Hot
```

FIGURE 7.17 The object diagram for the **HiLo** program.

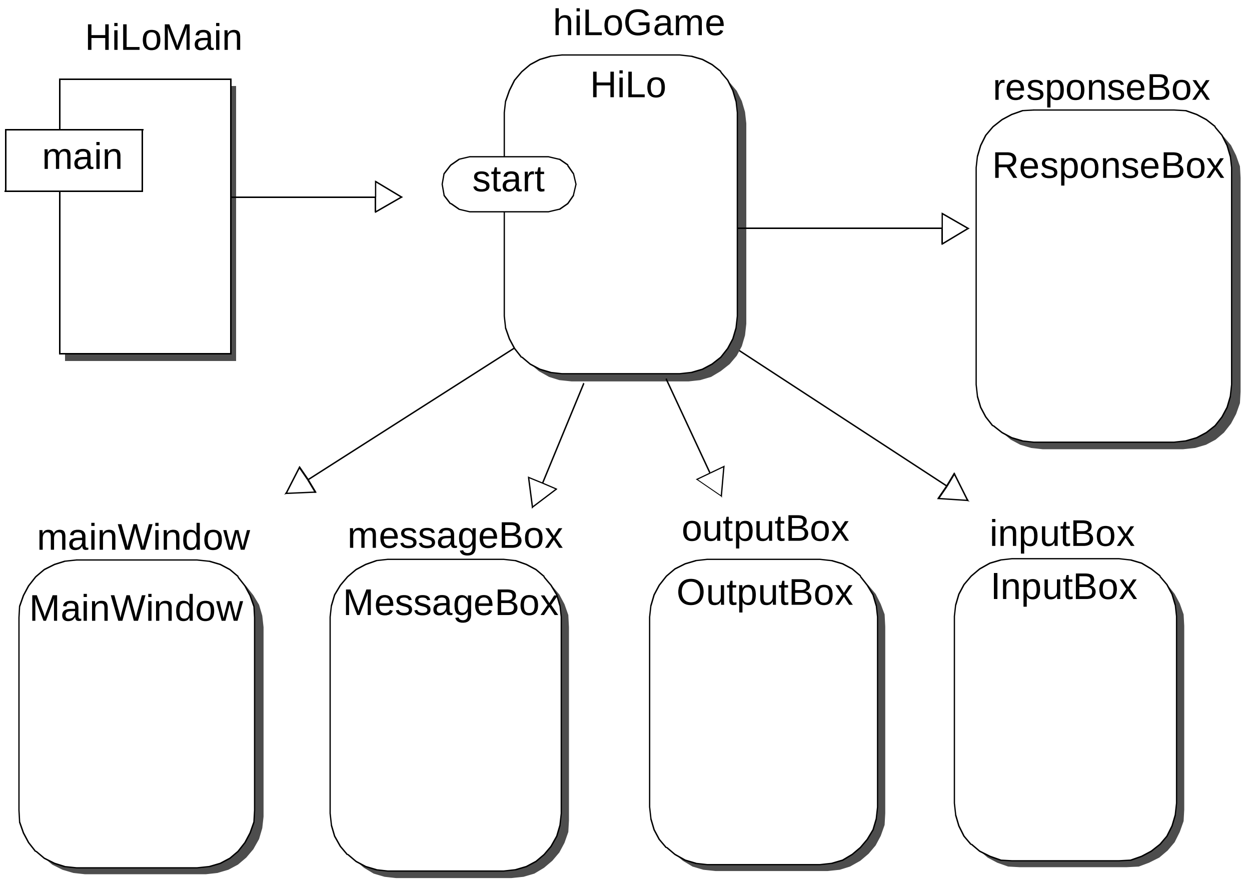
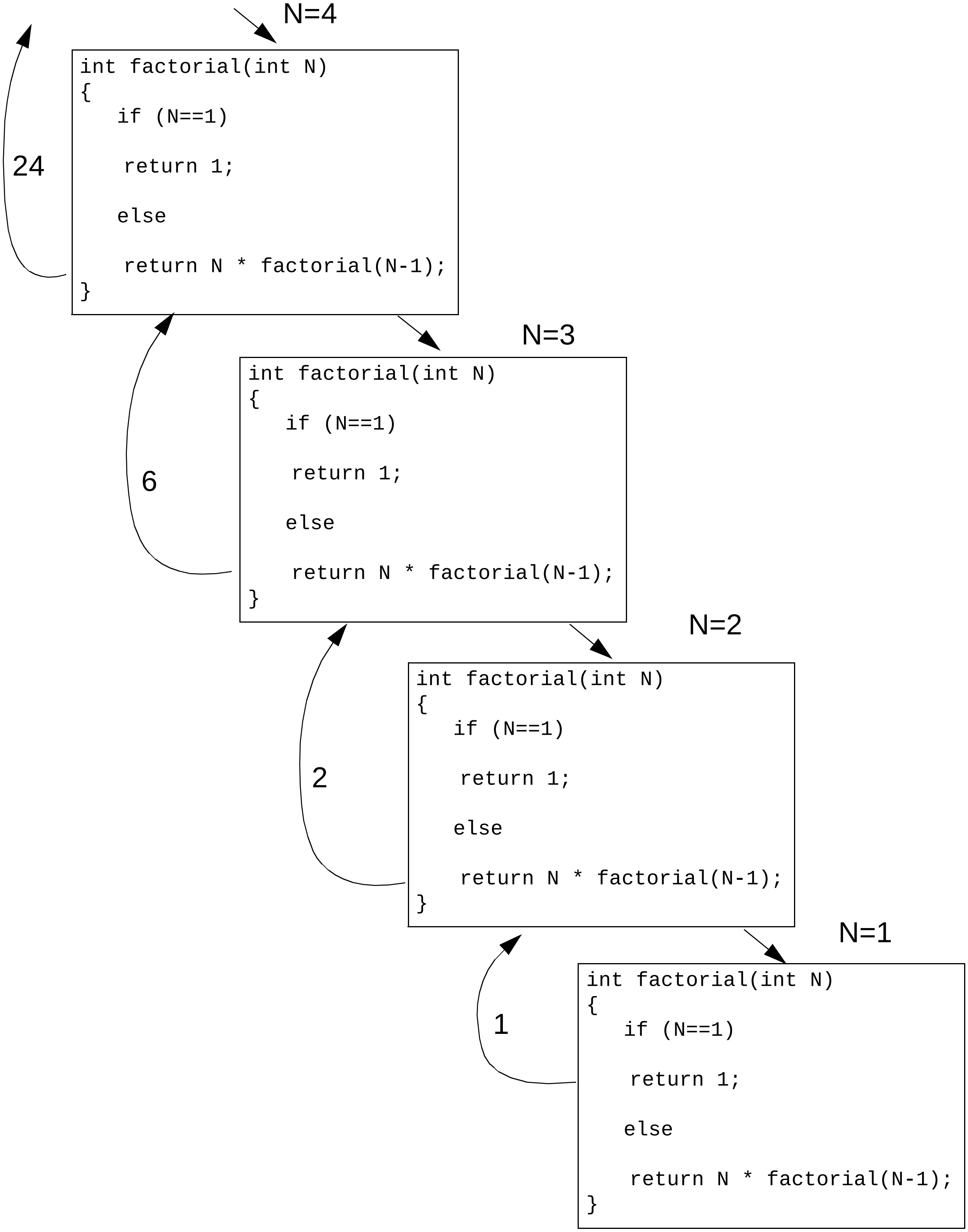


FIGURE 7.18 The sequence of calls for the recursive **factorial** method.



```
int factorial(int N)
{
   if (N==1)

    return 1;

   else

    return N * factorial(N-1);
}
```

```
int factorial(int N)
{
   if (N==1)

    return 1;

   else

    return N * factorial(N-1);
}
```

```
int factorial(int N)
{
   if (N==1)

    return 1;

   else

    return N * factorial(N-1);
}
```

```
int factorial(int N)
{
   if (N==1)

    return 1;

   else

    return N * factorial(N-1);
}
```