Conditions and Their Use for Controlling the Flow of Programs

6

6.1 Condition and Its Evaluation

A **condition** is a literal, a variable, a formula, or a method call whose value is `boolean`.[1] **Conditional evaluation** is the action of obtaining the value of a condition. Recall, as we studied in Sect. 2.2, that `boolean` is a primitive data type with just two possible values, `true` and `false`, which are opposite to each other. `System.out.print` and its variants print these two values as the `String` literals `"true"` and `"false"`, respectively, as demonstrated in the next code:

```
1  public class BooleanPrint
2  {
3    public static void main( String[] args )
4    {
5      boolean t = true;
6      boolean f = false;
7      System.out.println( t );
8      System.out.println( f );
9    }
10 }
```

Listing 6.1 An program that prints `boolean` literals and variables

The code produces the following result:

```
1 true
2 false
```

To build a `boolean` formula, **conditional** or **logical operators** can be used. There are three conditional operators, the **negation**, **conjunction** (or **logical-and**), and **disjunction** (or **logical-or**).

Negation The **negation** of a condition has the opposite value of the original; that is, if `x` has the value of `true`, `!x` has the value of `false`; and if `x` has the value of `false`, `!x` has the value of `true`. The negation operator must be attached immediately in front of the condition it acts upon.

[1] The name "boolean" comes from **George Boole** (November 2, 1815 to December 8, 1864), a nineteenth century English mathematician who did fundamental work in logic and algebra.

M. Ogihara, *Fundamentals of Java Programming*,
https://doi.org/10.1007/978-3-319-89491-1_6

The negation can be applied multiple times. When the negation is applied to something that is already negated, a pair of matching parentheses is needed before attaching the additional negation. In other words, for any condition `x`, the double negation `!!x` must be written as `!(!x)`. The double negation of a condition has the same value as the original.

Disjunction The disjunction asks whether or not at least one of the conditions given as the operands have the value of `true`. For operands `x1`, ..., `xk` such that `k` is greater than or equal to 2,

```
x1 || ... || xk
```

has the value of `true` if and only if for some `i`, `xi` has the value of `true`.

Conjunction The conjunction asks whether or not all of the conditions given as the operands have the value of `true`. For operands `x1`, ..., `xk` such that `k` is greater than equal to 2,

```
x1 && ... && xk
```

has the value of `true` if and only if for all `i`, `xi` has the value of `true`.

De Morgan's laws state[2]:

`!(x && y)` is equivalent to `!x || !y` and
`!(x || y)` is equivalent to `!x && !y`

Since the double negation flips the value back to the original, we have:

`x && y` is equivalent to `!(!x || !y)` and
`x || y` is equivalent to `!(!x && !y)`

The evaluation of conditional formulas with more than two operands follows rules analogous to the rules used in evaluating mathematical formulas, with `!`, `||`, and `&&` acting as `-`, `+`, and `*`, respectively.

The next code demonstrates the effect of the three `boolean` operations. In this program, we make a number of conditional evaluations. To make the code look simpler, we use the method `nameAndValue`. The method receives a `String` data `name` and a `boolean` data `value` as formal parameters and prints them in a single line with a `String` literal `": "` in between:

```
1 public class BooleanConnectivesNew
2 {
3   public static void nameAndValue( String name, boolean value )
4   {
5     System.out.println( name + ": " + value );
6   }
```

Listing 6.2 A program that demonstrates the use of `boolean` operators (part 1)

We use the method `nameAndValue` as follows. Line 9 of the program goes:

```
9     System.out.println( "---------- NOT --------------" );
```

Here, the first parameter `"true && true"` is a `String` literal that presents as a `String` literal a conditional formula to be made and the second parameter `true && true` is the actual formula to be evaluated. Note that the first actual parameter is a `String` literal and the second is a `boolean` formula.

The method `nameAndValue` simplifies the somewhat awkward single `println` statement of the form:

[2] Augustus De Morgan (27 June 1806 to 18 March 1871) was a British mathematician and logician, a contemporary of George Boole. He worked on logic and algebra.

```
    System.out.println( "true && true: " + (true && true) );
```

To execute this statement, `println` evaluates the conditional formula `(true && true)`. The value of the formula is `true`. After the evaluation, `println` converts the `boolean` value to a `String` literal `"true"`, and appends it to `"true && true: "`. As the results, `println` produces the output:

```
true && true: true
```

The method `main` that is shown next demonstrates how the three logical operators work by applying them to `boolean` literals:

```
1   public static void main( String[] args )
2   {
3     System.out.println( "---------- NOT --------------" );
4     nameAndValue( "!true", !true );
5     nameAndValue( "!false", !false );
6     nameAndValue( "!!true", !!true );
7     nameAndValue( "!!false", !!false );
8     System.out.println( "---------- AND --------------" );
9     nameAndValue( "true && true", true && true );
10    nameAndValue( "true && false", true && false );
11    nameAndValue( "false && true", false && true );
12    nameAndValue( "false && false", false && false );
13    System.out.println( "---------- OR ---------------" );
14    nameAndValue( "true || true", true || true );
15    nameAndValue( "true || false", true || false );
16    nameAndValue( "false || true", false || true );
17    nameAndValue( "false || false", false || false );
18  }
```

Listing 6.3 A program that demonstrates the use of `boolean` operators (part 2)

The execution of the code produces the following result:

```
1  ---------- NOT --------------
2  !true: false
3  !false: true
4  !!true: true
5  !!false: false
6  ---------- AND --------------
7  true && true: true
8  true && false: false
9  false && true: false
10 false && false: false
11 ---------- NO ---------------
12 true || true: true
13 true || false: true
14 false || true: true
15 false || false: false
```

Two conditions can be compared for equality and inequality. Given two conditions `x` and `y`, `x == y` tests whether or not the value of `x` is equal to the value of `y`, and `x != y` tests whether or not the value of `x` is not equal to the value of `y`. If `x` (or `y`) is a formula, it may be necessary to surround it with a pair of parentheses.

Numbers and `char` data can be compared for equality and inequality. Let `x` and `y` be data of some non-`boolean` primitive data types, where the data type of `x` may be different from the data type of `y`. We can apply six different comparisons to them:

- `x > y` tests whether or not the value of `x` is strictly greater than the value of `y`.
- `x >= y` tests whether or not the value of `x` is greater than or equal to the value of `y`.
- `x < y` tests whether or not the value of `x` is strictly less than the value of `y`.
- `x <= y` tests whether or not the value of `x` is less than or equal to the value of `y`.
- `x == y` tests whether or not the value of `x` is equal to the value of `y`.
- `x != y` tests whether or not the value of `x` is not equal to the value of `y`.

In the case where either `x` or `y` is `char`, the `char` type is treated as an unsigned 16-bit number. For this treatment, we use the character table called **Unicode**.[3] An important subset of the character set is the set of characters whose values are between 0 and 127. We call the subset the **ASCII Table**.[4]

The equality and inequality tests can be applied to object data types, e.g., `String` data, but the tests do not compare the contents of the objects, but the data locations. A special value for object type data is `null`. `null` means that the value is undefined. The following program:

```
1  boolean flag;
2  String unknown;
3  flag = ( unknown == null );
4  System.out.print( flag );
5  unknown = "abc";
6  flag = ( unknown == null );
7  System.out.print( flag );
```

produces two lines of output:

```
1 true
2 false
```

Many object types in Java offer a method specifically for comparisons. They are usually called `equals` and `compareTo`. We shall see such methods for the `String` type in Chap. 9.

In the next code, the program prompts the user to enter two integers, receives two numbers, and then performs the six comparisons between the two values entered. The program then executes the same series of action by receiving two real numbers from the user. The results of the six comparisons are stored in six `boolean` variables using the statement of the form:

```
BOOLEAN_VARIABLE = OPERAND1 OPERATOR OPERAND2;
```

For variable naming, the code uses the form `eqXXX`, `neXXX`, `gtXXX`, `geXXX`, `ltXXX`, and `leXXX` for `==`, `!=`, `>`, `>=`, `<`, and `<=`, with `Int` for `XXX` for integers and `Double` for floating point numbers.

Here is the header part of the program. It states what variables will be used:

[3]For example, `https://unicode-table.com/en/#control-character`.

[4]For example, `http://www.asciitable.com`.

```
1  import java.util.*;
2  // various comparisons
3  public class Comparisons0
4  {
5    public static void main( String[] args )
6    {
7      Scanner keyboard = new Scanner( System.in );
8      int int1, int2;
9      double double1, double2;
10     boolean eqInt, neInt, gtInt, geInt, ltInt, leInt;
11     boolean eqDouble, neDouble, gtDouble, geDouble, ltDouble, leDouble;
```

Listing 6.4 A program that shows comparisons between numbers (part 1)

In the next part, the program receives two `int` data from the user, compares them in six different ways, saves the outcomes in their respective variables, and prints the outcomes along with the numbers:

```
12     //------------- enter int values
13     System.out.print( "Enter two int: " );
14     int1 = keyboard.nextInt();
15     int2 = keyboard.nextInt();
16     //------------- compare the int values
17     eqInt = ( int1 == int2 );
18     neInt = ( int1 != int2 );
19     gtInt = ( int1 > int2 );
20     geInt = ( int1 >= int2 );
21     ltInt = ( int1 < int2 );
22     leInt = ( int1 <= int2 );
23     //------------- print the results
24     System.out.printf( "int1 = %d, int2 = %d%n", int1, int2 );
25     System.out.printf( "int1 == int2 returns %s%n", eqInt );
26     System.out.printf( "int1 != int2 returns %s%n", neInt );
27     System.out.printf( "int1 > int2 returns %s%n", gtInt );
28     System.out.printf( "int1 >= int2 returns %s%n", geInt );
29     System.out.printf( "int1 < int2 returns %s%n", ltInt );
30     System.out.printf( "int1 <= int2 returns %s%n", leInt );
```

Listing 6.5 A program that shows comparisons between numbers (part 2)

To print the outcome, we use `System.out.printf` that we saw earlier in Sect. 5.3.2. The method `printf` follows the syntax:

```
printf( FORMAT_STRING, PARAMETER1, ..., PARAMETERk );
```

where `k` represents the number of placeholders for data values appearing in `FORMAT_STRING`. `FORMAT_STRING` is a `String` data. We call the first parameter of `printf` the **formatting String**.

In `System.out.printf`, each data placeholder takes the form of `%XXXt`, where `t` is a letter that refers to the expected data type and `XXX` is a character sequence that specifies the way in which the data value is printed. In this book we will see five types for `t`: `c` for `char`, `d` for any whole number, `f` for any floating point number, `s` for `String` and `boolean`, and, much later, `e` for exponential representation. If the `XXX` part is empty, `System.out.printf` uses the default formatting for the type `t` (see Chap. 8).

The first `printf` statement in our code is:

```
System.out.printf( "int = %d, int2 = %d%n", int1, int2 );
```

The format `String` of the statement is `"int = %d, int2 = %d%n"`. Two placeholders appear in it. Both placeholders are `%d` and thus meant for printing integers. The `%n` is equivalent to `\n` and is for printing the newline character. Since two placeholders appear, the `printf` statement works if and only if exactly two whole numbers appear after the format `String`. In our case, those numbers are `int1` and `int2`. When printing the values, `printf` formats use the minimum number of characters required to print the values. Suppose the value of `int1` is 987 and the value of `int2` is −456. Then `printf` produces the output:

```
int = 987, int2 = -456
```

According to the rule, if the values are 10 and −98, respectively, the output is:

```
int = 10, int2 = -98
```

In the ensuing six lines, the placeholder is `%s` and `printf` substitutes each placeholder with the value of the `boolean` variable appearing in the statement.

The last part of the code uses `%f` for printing the `double` values:

```
31      //------------- enter double values
32      System.out.print( "Enter two floating point numbers: " );
33      double1 = keyboard.nextDouble();
34      double2 = keyboard.nextDouble();
35      //------------- compare the Double values
36      eqDouble = ( double1 == double2 );
37      neDouble = ( double1 != double2 );
38      gtDouble = ( double1 > double2 );
39      geDouble = ( double1 >= double2 );
40      ltDouble = ( double1 < double2 );
41      leDouble = ( double1 <= double2 );
42      //------------- print the results
43      System.out.printf( "double1 = %f, double2 = %f%n",
44          double1, double2 );
45      System.out.printf( "double1 == double2 returns %s%n", eqDouble );
46      System.out.printf( "double1 != double2 returns %s%n", neDouble );
47      System.out.printf( "double1 > double2 returns %s%n", gtDouble );
48      System.out.printf( "double1 >= double2 returns %s%n", geDouble );
49      System.out.printf( "double1 < double2 returns %s%n", ltDouble );
50      System.out.printf( "double1 <= double2 returns %s%n", leDouble );
51    }
52  }
```

Listing 6.6 A program that shows comparisons between numbers (part 3)

Here is one execution example of the program:

```
1  Enter two int: -9834 5343
2  int1 = -9834, int2 = 5343
3  int1 == int2 returns false
4  int1 != int2 returns true
5  int1 > int2 returns false
6  int1 >= int2 returns false
7  int1 < int2 returns true
8  int1 <= int2 returns true
9  Enter two floating point numbers: -194.5 -34.5
```

```
10 double1 = -194.500000, double2 = -34.500000
11 double1 == double2 returns false
12 double1 != double2 returns true
13 double1 > double2 returns false
14 double1 >= double2 returns false
15 double1 < double2 returns true
16 double1 <= double2 returns true
```

6.2 The If Statements

6.2.1 If

Using conditional evaluations, we can control the flow of programs. We can create a code that adjoins two sequences of statements, A and B, with the evaluation of a condition C, in the following format:

"if C then execute A; otherwise, execute B"

We call this **conditional execution**. All programming languages, including Java, have conditional executions. The standard expression of a conditional execution is the use of the keyword `if`, so we call it an **if-statement**. For the alternate action (which corresponds to "otherwise"), the accompanying keyword is `else`.

The primary structure for an if-statement in Java, with no action to perform in C, is:

```
1 if ( CONDITION )
2 {
3   STATEMENTS
4 }
5 AFTER_IF_PART
```

Here, the `CONDITION` is the condition to be evaluated and `AFTER_IF_PART` is the actions to be performed after completing the if-statement. If the evaluation of `CONDITION` produces the value `false`, the execution jumps to this part.

In other words, this code fragment is executed as follows:

- `CONDITION` produces `true`: `STATEMENTS` followed by `AFTER_IF_PART`.
- `CONDITION` produces `false`: `AFTER_IF_PART`.

We can draw a diagram that describes this action (see Fig. 6.1).

A special feature of if-statements is that the pair of curly brackets following the conditional evaluation can be omitted if there is *only one statement* in the block. While the ability to omit the curly brackets is convenient, the omission can lead to logical errors.

The next code is our first example of using if-statements. The program receives a temperature value from the user and makes a comment.

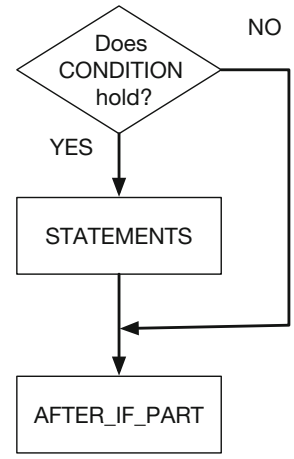

Fig. 6.1 The execution diagram of an if-statement

```
1  import java.util.Scanner;
2  // ask about temperature and respond
3  public class Temperature01
4  {
5    public static void main( String[] args )
6    {
7      Scanner keyboard = new Scanner( System.in );
8      //-- prompt answer
9      System.out.print( "What is the average high temperature in "
10       + "August in your area? : " );
11     double temp = keyboard.nextDouble();
12     //-- response
13     if ( temp > 90.0 )
14     {
15       System.out.println( "Wow! That must be very hot!" );
16     }
17   }
18 }
```

Listing 6.7 A program that receives a temperature value and makes a comment when appropriate

Here is the if-statement appearing the code:

```
1      if ( temp > 90.0 )
2      {
3        System.out.println( "Wow! That must be very hot!" );
4      }
```

The program prints the statement `"... very hot!"` if the value the user has entered is strictly greater than 90.0 and prints nothing otherwise.

Here are three separate executions of the program.

In the first round of execution, the value is 85 and strictly greater than 90.0, so no message appears, as shown next:

```
What is the average high temperature in August in your area? : 85
```

In the second round, the value is 90 and not strictly less than 90.0, so no message appears, as shown next:

```
What is the average high temperature in August in your area? : 90
```

In the third round, the value is 91 and is strictly greater than 90.0, so the message appears, as shown next:

```
1 What is the average high temperature in August in your area? : 91
2 Wow! That must be very hot!
```

Figure 6.2 shows the diagram of the program.

Our next code example uses two if-statements. The first if-statement tests if the value entered is strictly greater than 90.0 as before. The second one tests if the temperature value is less than or equal to 70.0 (notice the equality sign placed on the second one). There are two messages from which the program chooses to print. The program prints the first message if and only if the temperature is strictly greater than 90.0, and the second message if and only if the temperature is less than or equal to 70.0. The program prints no statement if the temperature is strictly greater than 70.0 and less than or equal to 90.0:

```
1  import java.util.Scanner;
2  // ask about temperature and respond
3  public class Temperature02
4  {
5    public static void main( String[] args )
6    {
7      Scanner keyboard = new Scanner( System.in );
8      //-- prompt answer
9      System.out.print( "What is the average high temperature in "
10         + "August in your area? : " );
11     double temp = keyboard.nextDouble();
12     //-- response no.1
13     if ( temp > 90.0 )
14     {
15       System.out.println( "Wow! That must be very hot!" );
16     }
17     //-- response no.2
18     if ( temp <= 70.0 )
19     {
20       System.out.println( "Wow! That must be very cold!" );
21     }
22   }
23 }
```

Listing 6.8 Another program that receives a temperature value and makes a comment when appropriate

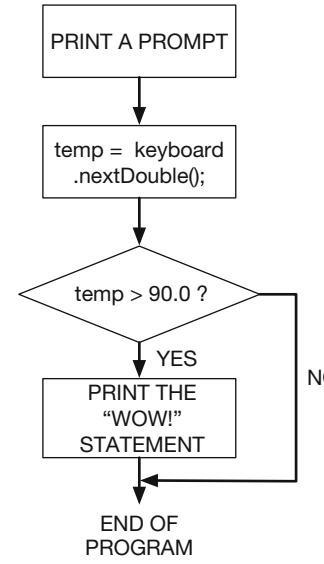

Fig. 6.2 The execution diagram of `Temperature01.java`

Here are three separate executions of the program:

```
What is the average high temperature in August in your area? : 60
Wow! That must be very cold!
```

```
What is the average high temperature in August in your area? : 80
```

```
% java Temperature02
What is the average high temperature in August in your area? : 91
Wow! That must be very hot!
```

In the next code example, the program asks the user to enter a temperature value and then a humidity value. The if-statements of the program combine the tests on the temperature and humidity values using the conjunction operator `&&`:

```
import java.util.Scanner;
// ask about temperature and humidity and provide response
public class Temperature03
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    //-- prompt answer
    System.out.print( "What is the average high temperature in "
      + "August in your area? : " );
    double temp = keyboard.nextDouble();
    System.out.print( "How about the average humidity? : " );
    double humidity = keyboard.nextDouble();
    //-- response no.1
    if ( temp >= 90.0 && humidity >= 90.0 )
    {
      System.out.println( "Wow! That must be hot and humid!" );
    }
    //-- response no.2
    if ( temp >= 90.0 && humidity <= 50.0 )
    {
      System.out.println( "Wow! That must be hot and dry!" );
    }
    //-- response no.3
    if ( temp <= 70.0 )
    {
      System.out.println( "Wow! That must be cool!" );
    }
    //-- response no.4
    if ( temp > 70.0 && humidity < 90.0 )
    {
      System.out.println( "Wow! That must be very comfortable!" );
    }
  }
}
```

Listing 6.9 A program that receives a temperature value and a humidity value, and then makes a comment when appropriate

Here are some execution examples:

```
What is the average high temperature in August in your area? : 80
How about the average humidity? : 100
```

```
What is the average high temperature in August in your area? : 70
How about the average humidity? : 50
Wow! That must be cool!
```

```
What is the average high temperature in August in your area? : 91
How about the average humidity? : 90
Wow! That must be hot and humid!
```

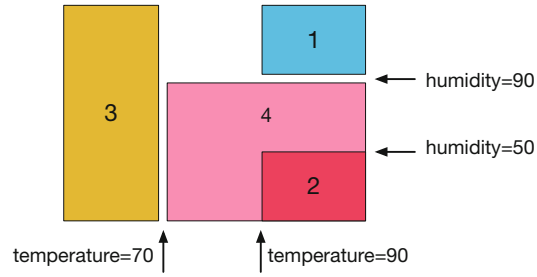

Fig. 6.3 The combinations of temperature and humidity considered in `Temperature03`

```
1 What is the average high temperature in August in your area? : 95
2 How about the average humidity? : 40
3 Wow! That must be hot and dry!
4 Wow! That must be very comfortable!
```

What are the cases in which the program produces no output? Also, what are the cases in which the program produces more than one statement?

The four cases the program tests are as follows:

Case 1 `temp >= 90.0 && humidity >= 90.0`
Case 2 `temp >= 90.0 && humidity <= 50.0`
Case 3 `temp <= 70.0`
Case 4 `temp > 70.0 && humidity < 90.0`

We can draw a diagram shown in Fig. 6.3 to discern these four cases with the temperature as the x-axis, the humidity as the y-axis, and the rectangles representing the cases.

We thus have:

- the program produces no comments if and only if the temperature is strictly greater than 70.0 and is strictly less than 90.0 and the humidity is greater than or equal to 90.0

and

- the program produces two comments if the temperature is strictly greater than 70.0 and the humidity is less than or equal to 50.0.

Here is one more example of the use of if-statements.

The program presents a list of four colors indexed 1, . . . , 4 to the user. It then asks the user to select a number that represents her favorite color. Upon receiving the input, the program produces a message based upon the choice that the user has made. The program stores the response from the user in an `int` variable, `answer`, by assigning the value that the `nextInt` method returns.

There are four if-statements in the code. In the order of appearance, they have the following roles.

1. `if ( answer < 1 || answer > 4 )` checks whether or not the user's selection is invalid—the selection has to be one of 1, 2, 3, and 4. This one thus checks whether or not the number is either (strictly less than 1) or (strictly greater than 4). If either is the case, the program produces a message that says the choice is invalid.
2. `if ( answer >= 1 && answer <= 4 )` tests the validity of the choice. It uses the condition exactly opposite to the first one.

3. `if ( answer == 1 || answer == 2 )` is for producing a special message when the choice corresponds to one of the University of Miami (UM) colors (orange and green).
4. `if ( answer == 3 || answer == 4 )` is for producing a special message when the choice corresponds to one of the University of Michigan (UM) colors (maize and blue).

Note that the messages that the first two if-statements generate use `printf` for formatting:

```
1  import java.util.Scanner;
2  // ask about a color and respond
3  public class ColorSelection
4  {
5    public static void main( String[] args )
6    {
7      //-- scanner
8      Scanner keyboard = new Scanner( System.in );
9      System.out.println( "What is your favorite color?" );
10     System.out.println(
11         "1. Orange, 2. Green, 3. Maize, 4. Blue" );
12     System.out.print( "Select from 1 to 4 : " );
13     int answer = keyboard.nextInt();
```

Listing 6.10 An interactive program that responds to the user's color selection (part 1)

```
14     if ( answer < 1 || answer > 4 )
15     {
16       System.out.printf( "Your choice %d is invalid.%n", answer );
17     }
18     if ( answer >= 1 && answer <= 4 )
19     {
20       System.out.printf( "Your choice %d is excellent.%n", answer );
21     }
22     if ( answer == 1 || answer == 2 )
23     {
24       System.out.println( "It is a U. Miami color!" );
25     }
26     if ( answer == 3 || answer == 4 )
27     {
28       System.out.println( "It is a U. Michigan color!" );
29     }
30   }
31 }
```

Listing 6.11 An interactive program that responds to the user's color selection (part 2)

Following is an example of executing the program:

```
1  What is your favorite color?
2  1. Orange, 2. Green, 3. Maize, 4. Blue
3  Select from 1 to 4 : 6
4  Your choice 6 is invalid.
```

```
1  What is your favorite color?
2  1. Orange, 2. Green, 3. Maize, 4. Blue
3  Select from 1 to 4 : 3
4  Your choice 3 is excellent.
5  It is a U. Michigan color!
```

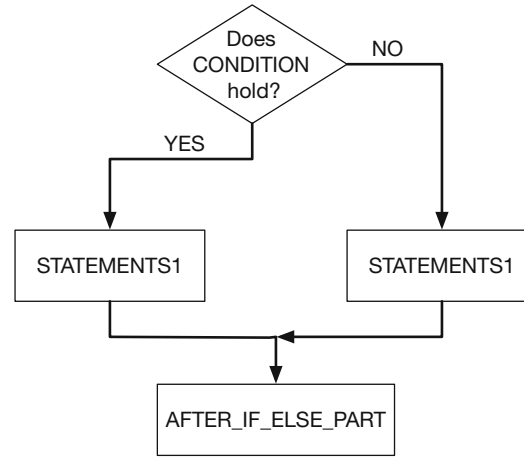

Fig. 6.4 The execution diagram of an if-else statement

```
What is your favorite color?
1. Orange, 2. Green, 3. Maize, 4. Blue
Select from 1 to 4 : 1
Your choice 1 is excellent.
It is a U. Miami color!
```

6.2.2 Else

Now we look at the if-statements having the "otherwise" part, which we call if-else statements:

```
if ( CONDITION )
{
  STATEMENTS1
}
else
{
  STATEMENTS2
}
AFTER_IF_ELSE_PART
```

Figure 6.4 shows the execution diagram of an if-else statement as it appears in the above hypothetical code.

In an if-else statement, an `if` and an `else` work as a pair. For each `if-else` pair, the `if` part must appear before the `else` part. Furthermore, the `if` part and the `else` part must be at the same depth. Furthermore, there must be no other statements or code blocks between the paired `if` and `else` parts.

The following example shows syntactically incorrect if-else statements:

```
public static void SOME_METHOD()
{
  else
  {
    ...
  }
  if ( CONDITION1 )
  {
```

```
    ...
  }
  System.out.println( "Wow!" );
  else
  {
    ...
  }
}
```

Similar to the case of if-statements, the curly brackets for the else-part can be omitted if there is only one statement in it. In other words,

```
if ( CONDITION )
  FIRST_STATEMENT;
else
  SECOND_STATEMENT;
AFTER_IF_ELSE_PART
```

is the same as:

```
if ( CONDITION )
{
  FIRST_STATEMENT;
}
else
{
  SECOND_STATEMENT;
}
AFTER_IF_ELSE_PART
```

The code:

```
if ( CONDITION )
  FIRST_STATEMENT;
  ADDITIONAL_STATEMENT;
else
  SECOND_STATEMENT;
```

is syntactically incorrect, because the code is basically saying:

```
if ( CONDITION )
{
  FIRST_STATEMENT;
}
  ADDITIONAL_STATEMENT;
else
{
  SECOND_STATEMENT;
}
```

meaning `ADDITIONAL_STATEMENT` is wedged between the if-part and the else-part.

Using `else` we can rewrite `ColorSelection` as follows:

```
import java.util.Scanner;
// ask about a color and respond
public class ColorSelectionElse
{
  public static void main( String[] args )
  {
    //-- scanner
    Scanner keyboard = new Scanner( System.in );
    System.out.println( "What is your favorite color?" );
    System.out.println(
        "1. Orange, 2. Green, 3. Maize, 4. Blue" );
    System.out.print( "Select from 1 to 4 : " );
    int answer = keyboard.nextInt();
    if ( answer < 1 || answer > 4 )
    {
      System.out.printf( "Your choice %d is invalid.%n", answer );
    }
    else
    {
      System.out.printf( "Your choice %d is excellent.%n", answer );
    }
    if ( answer == 1 || answer == 2 )
    {
      System.out.println( "It is a U. Miami color!" );
    }
    if ( answer == 3 || answer == 4 )
    {
      System.out.println( "It is a U. Michigan color!" );
    }
  }
}
```

Listing 6.12 A program that responds to a color selection of the user. The program uses `else`

6.2.3 If-Else Inside If/Else

Any number of if-statements and/or if-else statements can be placed in an `if`-block and in an `else`-block to build complex flow control.

The next code utilizes two `if-else` blocks, with the second one appearing inside the `else` block of the first, to accomplish exactly the same task as before:

```
import java.util.Scanner;
// ask about a color and respond
public class ColorSelectionInside
{
  public static void main( String[] args )
  {
    //-- scanner
    Scanner keyboard = new Scanner( System.in );
```

Listing 6.13 A color-selection program that uses two `if-else` blocks (part 1)

In the second occurrence of `if-else` (Lines 21–28), if the code reaches Line 20, the value of `answer` is guaranteed to be one of 1, 2, 3, and 4. Therefore if the code reaches the second `else` (Line 25), the value of `answer` is guaranteed to be either 3 or 4. This implies that the code works as we intended.

```
9       System.out.println( "What is your favorite color?" );
10      System.out.println(
11          "1. Orange, 2. Green, 3. Maize, 4. Blue" );
12      System.out.print( "Select from 1 to 4 : " );
13      int answer = keyboard.nextInt();
14      if ( answer < 1 || answer > 4 )
15      {
16        System.out.printf( "Your choice %d is invalid.%n", answer );
17      }
18      else
19      {
20        System.out.printf( "Your choice %d is excellent.%n", answer );
21        if ( answer == 1 || answer == 2 )
22        {
23          System.out.println( "It is a U. Miami color!" );
24        }
25        else
26        {
27          System.out.println( "It is a U. Michigan color!" );
28        }
29      }
30    }
31  }
```

Listing 6.14 A color-selection program that uses two `if-else` blocks (part 2)

A special case of successive `if-else` statements is `else if` as follows:

```
1  if ( CONDITION1 ) { STATEMENTS1 }
2  else if ( CONDITION2 ) { STATEMENTS2 }
3  else if ( CONDITION3 ) { STATEMENTS3 }
4  else { STATEMENTS4 }
```

Because of the rule allowing the omission of the curly-brackets after `else` (if the section has only one statement), and because a single-pair of `if-else` is an inseparable block of code, this is syntactically equivalent to:

```
1  if ( CONDITION1 ) { STATEMENTS1 }
2  else {
3    if ( CONDITION2 ) { STATEMENTS2 }
4    else { if ( CONDITION3 ) { STATEMENTS3 }
5      else { STATEMENTS4 }
6    }
7  }
```

In the above if-else statement, the evaluation of `CONDITION2` occurs only if the evaluation of `CONDITION1` produces `false` and the evaluation of `CONDITION3` occurs only if both the evaluation of `CONDITION1` and the evaluation of `CONDITION2` produces `false`. In general, if an if-statement is followed by a series of else-if statements, the condition evaluation terminates at the point where the result is `true` and evaluations beyond that point never take place. If there is no condition producing `true`, the statements corresponding to the final `else` will run.

Using this option, we can rewrite the previous code as follows:

```
1  import java.util.Scanner;
2  // ask about a color and respond
3  public class ColorSelectionWithElse
4  {
5    public static void main( String[] args )
6    {
7      //-- scanner
8      Scanner keyboard = new Scanner( System.in );
9      System.out.println( "What is your favorite color?" );
10     System.out.println( "1. Orange, 2. Green, 3. Yellow, 4. Blue" );
11     System.out.print( "Select from 1 to 4 : " );
12     int answer = keyboard.nextInt();
13     if ( answer < 1 || answer > 4 )
14     {
15       System.out.printf( "Your choice %d is invalid.%n", answer );
16     }
17     else if ( answer == 3 || answer == 4 )
18     {
19       System.out.printf( "Your choice %d is great, but%n", answer );
20       System.out.println( "it is not a UM color!" );
21     }
22     else
23     {
24       System.out.printf( "Your choice %d is great.%n", answer );
25     }
26   }
27 }
```

Listing 6.15 A color-selection program that uses `else-if`

The benefit of `else` is that when the flow control uses more than two mutually exclusive conditions, the expression can be simplified without having to spell out each condition succinctly.

To explain this a little, consider the hypothetical situation in Fig. 6.5. If we are to write the code without the use of `else`, it may look like:

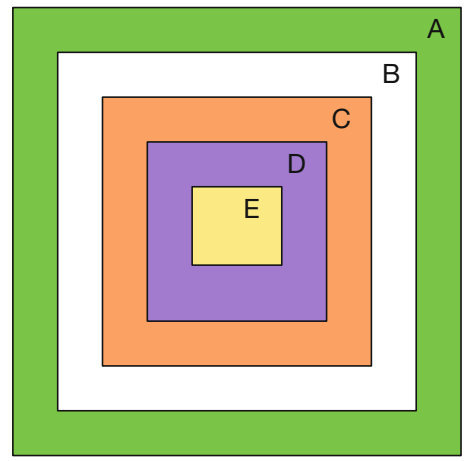

Fig. 6.5 A hypothetical situation with interwoven conditions

```
if ( cond_A ) { ...  }
if ( !cond_A )
{
  if ( cond_B ) { ...  }
  else ( !cond_B )
  {
    if ( cond_C ) { ...  }
    if ( !cond_C )
    {
      if ( cond_D ) { ...  }
      if ( !cond_D )
      {
        if ( cond_E ) { ...  }
        if ( !cond_E ) { ...  }
      }
    }
 }
}
```

or:

```
if ( cond_A ) { ...  }
if ( !cond_A && cond_B ) { ...  }
if ( !cond_A && !cond_B && cond_C ) { ...  }
if ( !cond_A && !cond_B && !cond_C && cond_D ) { ...  }
if ( !cond_A && !cond_B  && !cond_C && !cond_D && cond_E ) { ...  }
```

On the other hand, if we use `else`, we can simplify it as:

```
if ( cond_A ) { ...  }
else if ( cond_B ) { ...  }
else if ( cond_C ) { ...  }
else if ( cond_D ) { ...  }
else if ( cond_E ) { ...  }
```

Here is an example that demonstrates the benefit of `else`.

Suppose we are to write a program that receives an `int` value from the user and produces an output line depending on the value: message X for values 0, 4, 8, and 12, message Y for values less than 0 and values greater than 12, and message Z for the rest. Without using `else` the code may look like:

```
if ( value == 0 || value == 4 || value == 8 || value == 12 )
{
  System.out.println( X );
}
if ( value < 0 || value > 12 )
{
  System.out.println( Y );
}
if ( value > 0 && value < 4
  || value > 4 && value < 8
  || value > 8 && value < 12 )
{
  System.out.println( Z );
}
```

Using `else`, we can avoid using one conditional evaluation:

```
if ( value == 0 || value == 4 || value == 8 || value == 12 )
{
  System.out.println( X );
}
else if ( value < 0 || value > 12 )
{
  System.out.println( Y );
}
else
{
  System.out.println( Z );
}
```

By swapping the order between the first and the second conditions, we can further simplify the code, since 0, 4, 8, and 12 are all multiples of 4, as shown next:

```
if ( value < 0 || value > 12 )
{
  System.out.println( Y );
}
else if ( value % 4 == 0 )
{
  System.out.println( X );
}
else
{
  System.out.println( Z );
}
```

6.2.4 Truncation of Conditional Evaluations

The evaluations of a conditional formula generally proceeds from left to right and stops immediately when the value of the formula has been determined. For example, in the evaluation of a formula `( A || B || C )`, the evaluation order of the operands is `A`, `B`, and `C`. If the value of `A` is found to be `true`, regardless of the values of `B` and `C`, the value of the formula is `true`, so neither `B` nor `C` are evaluated. For a similar reason, if the value of `A` is found to be `false` and the value of `B` is found to be `true`, `C` is not evaluated. Similarly, in the condition `( A && B && C )`, if the value of `A` is found to be `false`, neither `B` nor `C` are evaluated, and if the value of `A` is found to be `true` and the value of `B` is found to be `false`, `C` is not evaluated.

This feature can be taken advantage of in many ways. Here is a simple example.

Suppose we are to write an application in which we receive two integers, `a` and `b`, from the user and test whether or not `a` is divisible by `b`. We can test the divisibility using the condition `a % b == 0`.

```
Scanner keyboard = new Scanner( System.in );
int a, b;
System.out.print( "Enter two integers a and b: " );
a = keyboard.nextInt();
b = keyboard.nextInt();
if ( a % b == 0 )
{
  System.out.println( a + " divides " + b );
}
```

With this code, if the user enters `0` for `b`, the execution results in a run-time error of `ArithmeticException`.

To prevent this error from happening, we can test `b != 0` first to ensure that the division is performed only if `b` is not 0:

```
1  Scanner keyboard = new Scanner( System.in );
2  int a, b;
3  System.out.print( "Enter two integers a and b: " );
4  a = keyboard.nextInt();
5  b = keyboard.nextInt();
6  if ( b != 0 )
7  {
8    if ( a % b == 0 )
9    {
10     System.out.println( a + " divides " + b );
11   }
12 }
```

We can apply the "truncation rule" to this code:

```
1  Scanner keyboard = new Scanner( System.in );
2  int a, b;
3  System.out.print( "Enter two integers a and b: " );
4  a = keyboard.nextInt();
5  b = keyboard.nextInt();
6  if ( b != 0 && a % b == 0 )
7  {
8    System.out.println( a + " divides " + b );
9  }
```

If `b == 0`, the conditional evaluation halts immediately with the outcome of `false`, so the remainder operator `a % b` will not occur. Thus, the program works the same way.

Another nice feature of conditional evaluation is that each of the six comparisons can be combined with an assignment. For example, consider the following code fragment:

```
1  int a, b
2  ...
3  if ( ( a = 2 * b ) > 17 )
4  {
5    ...
6  }
```

Here, `a` acquires the value of `2 * b`, and then the value is compared with 17.

In a little more complicated situation, an assignment of a value obtained by a `Scanner` object can be subjected to a test:

```
1  Scanner keyboard = new Scanner( System.in );
2  int a;
3  System.out.println( "Enter an integer: " );
4  if ( ( a = keyboard.nextInt() ) % 2 == 0 )
5  {
6    System.out.println( a + " is an even number." );
7  }
```

Summary

- A condition is a formula, a variable, a literal, or a method call that has a `boolean` value.
- Numbers, including `char`, can be compared using mathematical comparisons `x > y`, `x >= y`, `x < y`, `x <= y`, `x == y`, and `x != y`.
- `!`, `||`, and `&&` are `boolean` operators.
- `if` and `if-else` statements control the flow of the program.
- Multiple `else-if` blocks may appear after one `if`.
- The evaluation of a conditional formula involving multiple operands terminates as soon as the value of the formulas has been found.

Exercises

1. **Flattening multi-level if-then statements** Consider the following code:

```
1  if ( x > 10 )
2  {
3    if ( x > 20 )
4    {
5      methodA();
6    }
7    else
8    {
9      methodB();
10   }
11 }
12 else if ( x < -10 )
13 {
14   if ( x < -20 )
15   {
16     methodA();
17   }
18   else
19   {
20     methodB();
21   }
22 }
23 else
24 {
25   if ( x == 0 )
26   {
27     methodA();
28   }
29   else
30   {
31     methodB();
32   }
33 }
```

Rewrite the code in the form of:

```
if ( CONDITION_X )
{
  methodA();
}
else
{
  methodB();
}
```

2. **Flattening multi-level if-then statement, alternate**

```
if ( x > 10 )
{
  if ( x < 20 )
  {
    methodA();
  }
  else
  {
    methodB();
  }
}
else if ( x < -10 )
{
  if ( x > -20 )
  {
    methodA();
  }
  else
  {
    methodB();
  }
}
else
{
  if ( x == 0 )
  {
    methodA();
  }
  else
  {
    methodB();
  }
}
```

Rewrite the code in the form of:

```
if ( CONDITION_X )
{
  methodA();
}
else
{
  methodB();
}
```

3. **Tracing a nested if-statement** Consider the following code:

```
public static String test( int primary, int secondary )
{
  if ( primary > 0 )
  {
    if ( secondary > 0 )
    {
      return "1";
    }
  }
  return "0";
}
```

State:
- for which input combinations the method returns `"1"`;
- for which input combinations the method returns `"0"`.

4. **Tracing a nested if-statement** Consider the following code:

```
public static String test( int primary, int secondary )
{
  if ( primary > 0 )
  {
    if ( secondary == primary )
    {
      return "1";
    }
    else if ( secondary == 2 * primary )
    {
      return "2";
    }
    return "X";
  }
  return "0";
}
```

The return value of the method is `"1"`, `"2"`, `"X"`, or `"0"`. For each of the four possible return values, state one combination of the values of `primary` and `secondary` for which the method returns the value.

5. **Divisibility testing** Write a program, `Divisibility`, that receives two `int` values from the user and asserts whether the first number is a multiple of the second. The method asserts that the answer is in the negative if the second number happens to be 0.
6. **Boolean evaluation** Suppose `a`, `b`, and `c` are `boolean` variables. Then `( a == b ) != c` is a valid formula, since `a == b` evaluates to a `boolean`. There are eight possible combinations for the values of the three variables. For each of the combinations, state the value of the condition.
7. **Discriminant of a quadratic formula** Write a program, `DiscriminantTest`, that receives three real values `a`, `b`, and `c` from the user, and returns the number of real solutions of the equation $ax^2 + bx + c = 0$ as an `int`.
8. **Following the code execution to determine the output** State the value that the method `method123` below returns with each of the values below as the actual parameter:

```
public static double method123( double input )
{
  if ( input < 11.0 )
  {
    return input + 1;
  }
  if ( input < 12.0 )
  {
    return input + 2;
  }
  return input - 5;
}
```

(a) `10.0`
(b) `11.0`
(c) `12.0`
(d) `13.0`

9. **Following the code execution to determine the output** State the value the method `methodABC` below returns with each of the values below as the actual parameter:

```
public static double methodABC( int input )
{
  if ( input == 10 )
  {
    return input * 2;
  }
  if ( input < 11 )
  {
    return input * 3;
  }
  return input * 7;
}
```

(a) `9`
(b) `10`
(c) `11`

10. **Return value determination 1** Let a method `cute` be defined as follows:

```
public static boolean cute( boolean x, boolean y, boolean z )
{
  return x == (y == z);
}
```

For each of the eight possible input values, state the value the method returns.

11. **Return value determination 2** Let the method `neat` be defined as follows:

```
public static boolean neat( boolean x, boolean y, boolean z )
{
  return x != y || y != z || z != x;
}
```

State, for each of the eight possible value combinations for the three parameters, the return value of the method.

Programming Projects

12. **Triangle validity** Write a method, `isValidTriangle`, that receives three `double` values `sideA`, `sideB`, and `sideC` as parameters and returns whether or not each value is strictly less than the sum of the other two. The return type must be `boolean`.
13. **Right angle** Write a method, `isRightAngleTriangle`, that receives three `double` values `sideA`, `sideB`, and `sideC` as parameters, and returns whether or not the three edges form a right-angled triangle. The return type must be `boolean`.
14. **Checking whether or not three values are all positive** Write a public static method named `allPositive` that receives three `double` values `valueA`, `valueB`, and `valueC` as parameters, and returns whether or not the three values are all strictly positive. The return type must be `boolean`.
15. **What does this function compute?** Analyze the following code, and state what this method computes.

```
1 public static int mystery ( int a, int b, int c )
2 {
3   if ( a == Math.max( a, Math.max( b, c ) ) )
4   {
5     return Math.max( b, c );
6   }
7   return Math.max( a, Math.min( b, c ) );
8 }
```

16. **Solving a system of linear equations with two unknowns** Consider solving the system of linear equations with two unknowns x and y:

$$ax + by = s$$
$$cx + dy = t$$

To solve the problem, we first check the value of the determinant of the system: $h = ad - bc$. If h is not 0, we have $x = (ds - bt)/h$ and $y = (-cs + at)/h$. If h is 0, the system is degenerate, and falls into one of the following four cases:

- the system is unsolvable;
- the system is equivalent to one linear equation;
- the value of x is arbitrary and the value of y is fixed;
- the value of y is arbitrary and the value of x is fixed

Write a program, `LinearEquation2`, that receives the six coefficients from the user and solves the problem if the determinant h is not 0. The program should state, in the case that $h = 0$, that it cannot solve the system.

17. **Fully solving a system of linear equations with two unknowns** Continuing on the previous problem, write a program `LinearEquation2Full` that completely solves the problem including the cases in which only x or only y is determined. Here are five examples that show the behavior of the program.

```
1 Enter a, b, s for number 1: 1 2 3
2 Enter c, d, t for number 2: 3 1 2
3 x = 0.2, y = 1.4.
```

```
Enter a, b, s for number 1: 0 0 0
Enter c, d, t for number 2: 0 0 0
Both x and y are arbitrary.
```

```
Enter a, b, s for number 1: 1 0 2
Enter c, d, t for number 2: 2 0 4
y is arbitrary and x = 2.0.
```

```
Enter a, b, s for number 1: 1 2 3
Enter c, d, t for number 2: 2 4 6
Any point on 1.0x+2.0y=3.0.
```

```
Enter a, b, s for number 1: 1 2 3
Enter c, d, t for number 2: 2 4 5
Unsolvable.
```

Implement a series of cases to consider, which can be expressed as a series of if-else-if:

- "if" $h \neq 0$: solve it as before;
- "else if" either $a = b = 0$ and $s \neq 0$ or $c = d = 0$ and $t \neq 0$: unsolvable;
- "else if" either $a = b = s = c = d = t = 0$: arbitrary x and y;
- "else if" $a = b = s = 0$: equivalent to $cx + dy = t$;
- "else if" $c = d = t = 0$: equivalent to $ax + by = s$;
- "else if" $a = c = 0$ and $s/b = t/c$: equivalent to $ax + by = s$.
- "else if" $b = d = 0$ and $s/b = t/c$: equivalent to $ax + by = s$.
- "else if" either $a = c = 0$ or $b = d = 0$: unsolvable;
- "else if" $s/a \neq t/b$: unsolvable;
- "else": equivalent to $ax + by = s$.

Write a method named `justOne( double p, double q, double r )` that handles the situation in which the system is equivalent to just one equation, where it is guaranteed that either $p \neq 0$ or $q \neq 0$. Using this method, the series of actions can be handled in a slightly simpler manner.