COMP 110

Prasun Dewan[1]

12. Conditionals

Real-life algorithms seldom do the same thing each time they are executed. For instance, our plan for studying this chapter may be to read it in the park, if it is a sunny day, and in the library, otherwise. Similarly, the execution of most non-trivial programs depends on the results of tests. We will study new statements, called conditionals, that allow us to write such programs. They allow us to break away from the "straight-line" execution of the programs we have seen so far (where statements are executed one after the other along a linear execution path); in particular, they allow branching in the execution path. Because of the real-life analogy, using conditionals is relatively straightforward. The main challenge is to use them elegantly; therefore, we will also learn several general rules on their usage.

if-else statement

Suppose we need a procedure, `printPassFailStatus`, that takes a numeric score as an argument, and prints the string "PASS" or "FAIL" based on whether the score is above the pass cutoff. Thus, assuming the cutoff is 50, the call

```
printPassFailStatus(95)
```

should print:

```
PASS
```

and:

```
printPassFailStatus(30)
```

should print:

```
FAIL
```

In order to write this method, we need a mechanism that allows the execution of different statements based on some condition. The *if-else* statement, illustrated below, is one mechanism for supporting such conditional execution:

```
public static void printPassFailStatus(int score) {
      if (score < PASS_CUTOFF)
            System.out.println("FAIL");
      else
            System.out.println("PASS");
}
```

[1] © Copyright Prasun Dewan, 2000.

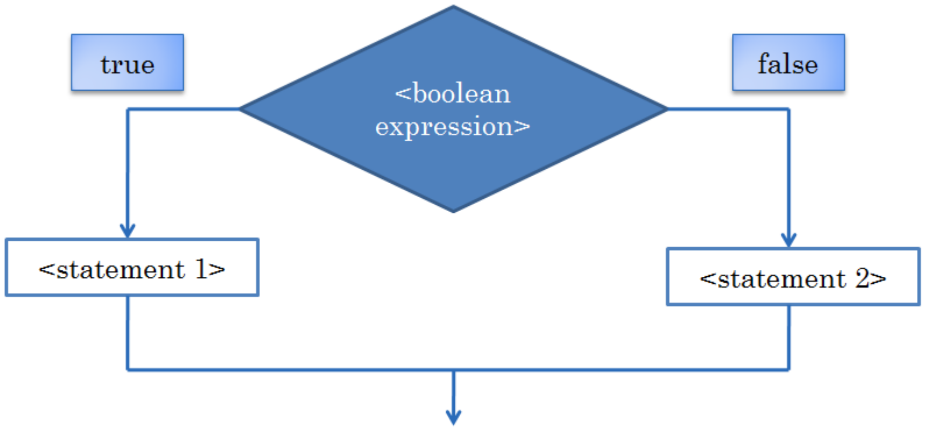


Figure 1. if-else statement

In general, the statement has the syntax:

```
if (<boolean expression>)
      <statement1>;
else
      <statement2>;
```

(Note that parentheses are required around `<boolean expression>`). What this statement depends on `<boolean expression>`. If `<boolean expression>` evaluates to **true**, it executes `<statement1>`, otherwise it executes `<statement2>`. `<statement 1>` is called the *then part* or *then branch*; `<statement 2>` is called the *else part* or *else branch* ; and `<boolean expression>` is called the *condition* of the if-else.

Thus, in the method above, the if-else tests the `boolean` expression:

```
score < PASS_CUTOFF.
```

In case this expression returns true, the statement branches to the then part:

```
System.out.println("FAILED");
```

Otherwise the statement branches to the else part:

```
System.out.println("PASSED");
```

When the method is called with the argument, 30, the `boolean` expression evaluates to true, resulting in the then part being executed; and when it is called with the argument 30, it evaluates to false, resulting in the else part being executed.

Reading if-else

Novice programmers sometimes add a redundant test in the condition of an if-else:

```
if (score < PASS_CUTOFF == true)
```

This expression perhaps reads better in English if we read:

```
if (<boolean expression>)
```

as it is written. However, it should be really read as:

```
if (<boolean expression>) is true
```

because, as mentioned above, the if-else evaluates the value of its condition and executes the then-part if the value is true. Thus, the above if-else should be read as:

```
if (score < PASS_CUTOFF == true) is true
```

Now it does not read well. It is equivalent but more long-winded than saying:

```
if (score < PASS_CUTOFF ) is true
```

In general, avoid testing a `boolean` expression for equality or inequality with the true and false values.

Boolean Expression vs. if-else

Let us consider a variation of the method above that, instead of printing a string, returns a boolean value to indicate the pass/fail status. We might be tempted to structure this function like the procedure:

```
public static boolean hasFailed(int score) {
	if (score < PASS_CUTOFF)
		return true;
	else
		return false;
}
```

The if-else we use here is similar to the one we used in the previous example except that its then and else parts return the false and true values, rather than printing the strings “FAILED” and “PASSED”, respectively. There is a simpler way to write this function:

```
public static boolean hasFailed(int score) {
	return (score < PASS_CUTOFF);
}
```

The function returns **true** if the boolean expression

```
score < PASS_CUTOFF
```

is **true** and **false** otherwise, which is exactly what our previous version did. It is easy to detect that the if-else above is unnecessary if we read it as:

```
if (score < PASS_CUTOFF) is true
	return true;
else
	return false;
```

The statement is essentially returning the value of its condition. Since an if-else can implement more general logic than a `boolean` expression, we are often tempted to use them for all logic. Therefore, once you have written an if-else, check if a simple `boolean` expression would have sufficed instead. If all the then and else branches do is compute the true (false) and false (true) values, respectively, then get rid of the if-else and use (the negation of the) condition as the computed value.

Compound Statement

What if we wanted to perform more than one statement in the then or else branch? Java lets us create a *compound statement* by enclosing a series of statements within the delimiters { and }, as shown below:

```
public void fancyPrintGrade(int score) {
      if (score < PASS_CUTOFF )
      {
            System.out.println("**************");
            System.out.println("FAILED");
            System.out.println("**************");
      } else {
            System.out.println("**************");
            System.out.println("PASSED");
            System.out.println("Congratulations!");
            System.out.println("**************");
            }
}
```

As far as the if-else is concerned, there is only one statement in the then or the else part. The statement happens to be a compound statement containing a list of statements. We have actually already seen examples of a compound statement – the body of a method and the try and catch block. We will see later the use of the compound statement in other constructs. A compound statement is also called a *statement list, statement block* or simply *block.*

Putting the delimiters ({ and }) of a compound statement on their own lines takes too much space; therefore the convention is to put them on the lines containing the if and else keywords, as shown below:

```
public void fancyPrintGrade(int score) {
      if (score < PASS_CUTOFF) {
            System.out.println("**************");
            System.out.println("FAILED");
            System.out.println("**************");
      } else {
            System.out.println("**************");
            System.out.println("PASSED");
            System.out.println("Congratulations!");
            System.out.println("**************");
      }
}
```

Code Duplication Again

It is possible to write a better implementation of the procedure above:

```
public void factoredFancyPrintGrade(int score) {
      System.out.println("**************");
      if (score < PASS_CUTOFF) {
            System.out.println("FAILED");
      } else {
            System.out.println("PASSED");
            System.out.println("Congratulations!");
      }
      System.out.println("**************");
}
```

While the previous implementation indicates the exact set of steps to be taken in each case, it repeats code in the two branches. It is preferable to share code that must be executed in both branches, for the reasons we saw earlier.

- *Less Typing*: We have to type less. With today's cut and paste features, this is not the major advantage, however, as we saw before.
- *Clarity:* We can clearly identify the code that is executed in both cases. Otherwise we would have to explicitly compare the two branches to see what code is executed in both cases. For instance, in the example above, we would have to count the number of *'s to se if the same number of them are being printed in the two cases.
- *Change:* Suppose we wanted to change the number of *'s that must be printed in both cases. In the non-sharing approach, we would have to duplicate the changes in both branches. It is easy to forget to change one of them or change them differently. So the sharing approach is more resilient to program changes.

The general principle, which we saw earlier, is to avoid duplication of code in a program. In the context of an if-else, it implies that we should share code in the then and else branches. We will later see the application of this principle to other statements.

if Statement

Sometimes we wish to take no action in an else branch. In this case, the *if statement* can be used instead of the if-else statement. It has the syntax:

```
if (<boolean expression>)
      <statement>;
```

Example:

```
if (score == MAX_SCORE)
      System.out.println ("Perfect Score! Congratulations!");
```

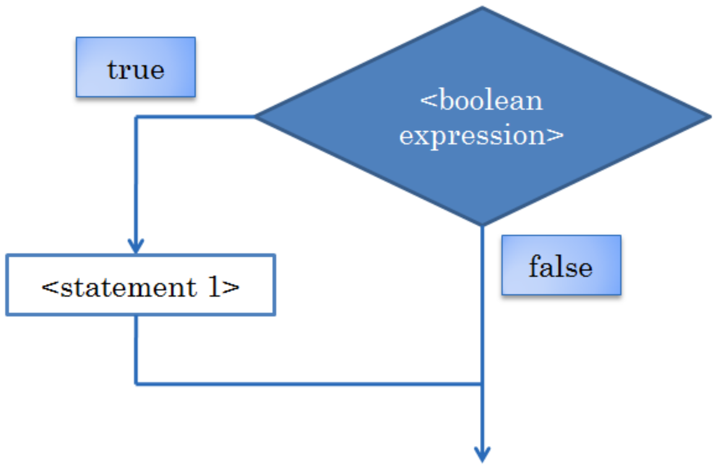


Figure 2. if statement

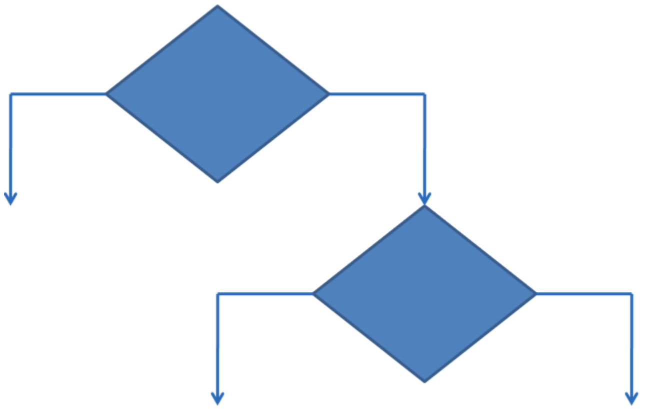


Figure 3. Nesting in the else parts

We can always use an if-else statement instead of an if statement by putting nothing (the "null" statement) in the else part:

```
if (score == MAX_SCORE)
      System.out.println ("Perfect Score! Congratulations!");
else;
```

But the if statement is more elegant when the else is a null statement. Beginning programmers, used mainly to the if-else statement and unfamiliar with the null statement, tend to sometimes put a spurious assignment in the else part:

```
if (score == MAX_SCORE)
      System.out.println ("Perfect Score! Congratulations!");
else
      score = score;
```

Like the previous solution, this is correct but even less elegant!

We shall refer to if-else and if statements as *conditionals*, because their execution depends on some condition. An if-else statement is used to choose between the execution of two statements based on the condition; while an if statement is used to determine whether a statement should be executed based on the condition.

Nested Conditionals

Consider the following function, which takes returns a letter grade based on the argument `score`:

```
public static char toLetterGrade(int score) {
      if (score >= A_CUTOFF)
            return 'A';
      else if (score >= B_CUTOFF)
            return 'B';
      else if (score >= C_CUTOFF)
            return 'C';
      else if (score >= D_CUTOFF)
            return 'D';
      else
            return 'F';
}
```

The use of the if else statement here might seem at odds with the syntax we gave:

```
if (<boolean expression>)
      <statement1>
else
      <statement2>
```

In fact, you might think of it as a special kind of statement in which the keywords **else if** replace the keyword **else**. But actually, it is simply a sequence of *nested conditionals*, that is, if-else statements whose else or then branches contains other conditionals (Figure 3). The following format clarifies this:

```
if (score >= A_CUTOFF)
      return 'A';
else
      if (score >= B_CUTOFF)
            return 'B';
      else
            if (score >= C_CUTOFF)
                  return 'C';
            else
                  if (score >= D_CUTOFF)
                        return 'D';
                  else
                        return 'F';
```

Since this format takes more space, we prefer the previous one, in which we used the same indentation for the enclosing and enclosed conditionals**.**

then vs. else Branching

We could have written this function in other several other ways. In the implementation above, all the branching (nesting) occurs in the else parts of the if-else statements (Figure 3). This is because we started our comparisons with the highest cutoff, A_CUTOFF. If we started our comparisons with the lowest one, D_CUTOFF, all the branching would occur in the then parts.

```
if (score >= D_CUTOFF)
      if (score >= C_CUTOFF)
            if (score >= B_CUTOFF)
                  if (score >= A_CUTOFF)
                        return 'A';
                  else
                        return 'B';
            else // score < B_CUTOFF
                  return 'C';
      else // score < C_CUTOFF
            return 'D';
else // score < D_CUTOFF
      return 'F';
```

However, this implementation is more awkward to read since the else parts of the inner if-else statements get separated from their conditions. In fact, we need comments to remind ourselves of the correspondence between these else part and their conditions. Therefore, it is better to nest the else branches rather than the then branches.

Linear Vs Balanced Branching

Both solutions, the branching is *linear*, that is, it occurs on one side (then or else) of the decision tree, with the other side not containing any conditional. If we started our comparisons, not with the extreme values, A_CUTOFF and D_CUTOFF, but instead with a value in between, we would get a more *balanced* branching:

```
if (score >= B_CUTOFF)
      if (score >= A_CUTOFF)
            return 'A';
      else
            return 'B';
else // score < B_CUTOFF
      if (score < C_CUTOFF)
            if (score < D_CUTOFF)
                  return 'F';
            else
                  return 'D';
      else // score >= C_CUTOFF
            return 'C';
```

Again, some else parts get separated from their conditions in this solution. Moreover, the solution is confusing because sometimes we check if a value is greater than a cutoff and sometimes if it is less. In summary, choose linear branching over balanced branching; and branching on the else side over branching on the then side.

Nested Vs Non-nested Solution

We could have done the problem without nested conditionals, as shown below:

```
if (score >= A_CUTOFF)
      result = 'A';
if ((score < A_CUTOFF) && (score >= B_CUTOFF))
      result = 'B';
if ((score < B_CUTOFF) && (score >= C_CUTOFF))
      result = 'C';
if ((score < C_CUTOFF) && (score >= D_CUTOFF))
      result = 'D';
if (score < D_CUTOFF)
      result = F';
return result;
```

The problem with this alternative is that there are two tests in the second, third, and fourth ifs, while there is only one test in the previous version for each if. Thus the previous version is more efficient - it will run faster. Moreover, you had to write less too! The second solution essentially violates the principle of avoiding redundant computation.

Multiple & Early Returns

As the various nested implementations of the function `toLetterGrade` how, it is possible for a function to have more than one return statement. When there are multiple alternative paths through a function, it is usual to have a return at the end of each path.

It is also possible to have an *early return*, that is, a return in the middle of a path. When Java encounters a return statement, it terminates execution of the method, ignoring any statements that follow it. An early return can be a useful feature, as illustrated by the following alternative solution to the letter grade problem. It uses independent if statements without doing redundant computation:

```
if (score >= A_CUTOFF)
      return 'A';
if (score >= B_CUTOFF)
      return 'B';
if (score >= C_CUTOFF)
      return 'C';
if (score >= D_CUTOFF)
      return 'D';
return ‘F';
```

This solution works because the then part of each of the if statements is a return statement, telling the method to ignore the following statements in the program. When we execute an if statement in this solution, we know that the condition of the previous if statement was false, which is equivalent to putting it in the else part of the preceding if statement. Thus, this program is equivalent to our first solution:

```
        if (score >= A_CUTOFF)
                return 'A';
        else if (score >= B_CUTOFF)
                return 'B';
        else if (score >= C_CUTOFF)
                return 'C';
        else if (score >= D_CUTOFF)
                return 'D';
        else
                return F';
```

It is somewhat preferable than the first solution since it requires a bit less typing.

In summary, a nested if then statement with multiple returns can sometimes be elegantly replaced by a series of if statements with early returns.

Explicit return in a Procedure

Java allows us to do an early return from a procedure also:

```
public void printLetterGrade(int score) {
        if (score < 0) {
                System.out.println ("Illegal score");
                return;
        }
        System.out.println("Letter Grade:" + toLetterGrade(score"));
}
```

In a procedure, the `return` keyword does not take an argument, because no value is to be returned. It is usual to have early returns in a procedure for various error conditions. In the absence of early returns, the normal (error-free) processing of the procedure would be in an else part of an if-else:

```
public void printLetterGrade(int score) {
        if (score < 0) {
                System.out.println ("Illegal score");
                return;
        else
                System.out.println(
                        "Letter Grade:" + toLetterGrade(score"));
}
```

This solution requires an extra level of indentation, which is a significant disadvantage because the screen real-estate tends to be limited.

Dangling Else Problem

Consider the following nested if statement:

```
        if (score >= B_CUTOFF)
```

```
if (score >= A_CUTOFF)
      System.out.println ("excellent!");
else
      System.out.println ("bad");
```

Which if statement does the else statement match with? If you typed this text using the J++ 6.0 editor, this is how it would be automatically formatted. This may lead you to believe that it matches the outer if statement. But you could manually format it as:

```
if (score >= B_CUTOFF)
      if (score >= A_CUTOFF)
            System.out.println ("excellent!");
      else
            System.out.println ("bad");
```

Now it looks as if it matches the inner if statement. Our syntax rule of an if-else statement allows for both interpretations. This ambiguity is called the *dangling else* problem. It is resolved by making an else match the closest if. So, in this example, the second interpretation is the correct one. The possibility for confusion here is another reason for not nesting in a then branch.

Summary

- An if-else conditional is used to choose between the execution of two statements based on the value of a boolean expression.
- An if conditional is used to determine whether a statement should be executed based on the value of a boolean expression.
- A conditional can nest other conditionals to create an arbitrary number of alternative execution paths within a method.
- In a nested conditional, branching in the else part is preferred to branching in the then part or balanced branching.
- An else part matches the closest then part.
- A nested if-then conditional with multiple returns can sometimes be elegantly replaced by a series of if statements with early returns.
- An if-else statement can sometimes be elegantly and efficiently replaced by a boolean expression.
- Avoid testing a boolean expression for equality or inequality with the `true` and `false` values.

Exercises

1. Write a more concise version of the following function:

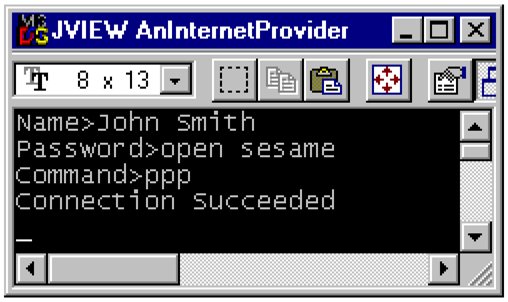
```
public static boolean hasPassed(int score) {
      if (score < PASS_CUTOFF)
            return false;
      else
            return true;
}
```

2. The following code fragment has repeated code in the two branches of the if-else statement. Write an equivalent function that does not have repeated code.

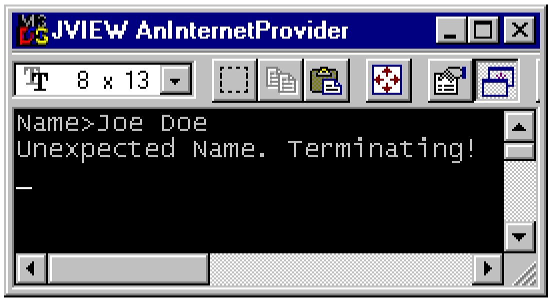
```
public void fancyPrintGrade(int score) {
      if (score < PASS_CUTOFF == true) {
            System.out.println("FAILED");
            System.out.println("**************");
            System.out.println("**************");
            System.out.println("Need to take the course again.");

      } else {
            System.out.println("PASSED");
            System.out.println("**************");
            System.out.println("**************");
            System.out.println("Congratulations!");
      }
}
```

3. Write a procedure, `internetLogin` to simulate process of manually logging in to an internet provider. The procedure takes as arguments the expected name and password of the user. It prompts the user for a name and password and checks the values input are consistent with the expected values. It then prompts the user for a command, and if the string “ppp” is entered, it outputs the string “ Connection Succeeded,” and returns to its caller. In case the wrong user name, password, or command are entered, it gives an error message indicating the erroneous input, and returns to its caller. The following window illustrates two possible interactions during calls to `internetLogin(“John Smith”, “open sesame”)`. Write the procedure using:
   a) branching in the else parts of nested if-else statements**.**
   b) branching in the then parts of nested if-else statements**.**
   c) a series of if statements with early returns.
   d) a single, non-nested, if-else statement that calls an auxiliary boolean function, `readAndCheckInput` that takes as arguments the kind of user input (“Name”, “Password”, “Command”) and the expected value for the entry, prompts the user for the input, reads the input, and returns true or false based on whether the actual user input is the same as the expected value.



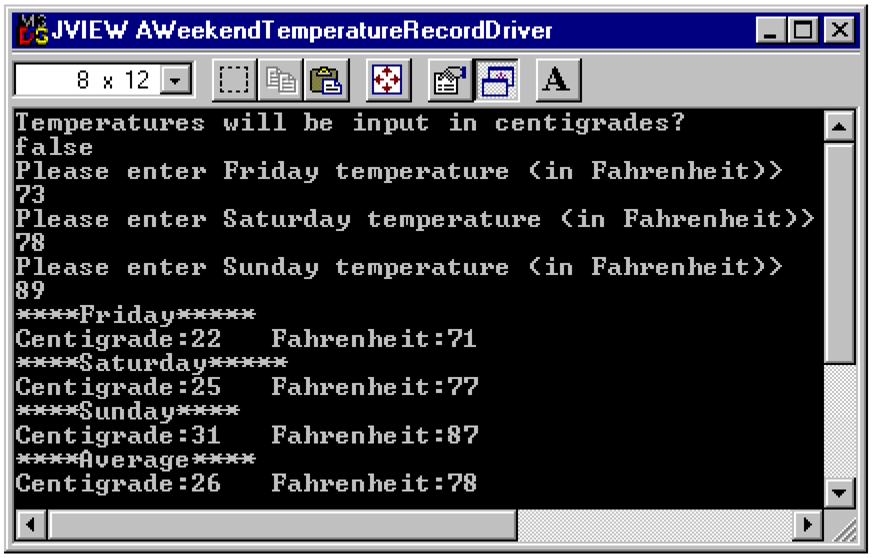
```
Name>John Smith
Password>open sesame
Command>ppp
Connection Succeeded
_
```



```
Name>Joe Doe
Unexpected Name. Terminating!
_
```

Check for string equality by invoking the `equals` method on a string, rather than using the Java == operator. Thus, to check that a string `s1` is the same as string `s2`, evaluate `s1.equals(s2)` rather than `s1==s2`. See chapter 17 for the difference between == and `equals`.

4. Extend the temperature driver class of Chapter 7, Question 4. to allow the user to enter the three temperatures in centigrade or Fahrenheit, depending on what the user chooses at the beginning of the interaction, as shown below:



```
Temperatures will be input in centigrades?
false
Please enter Friday temperature (in Fahrenheit))
73
Please enter Saturday temperature (in Fahrenheit))
78
Please enter Sunday temperature (in Fahrenheit))
89
****Friday*****
Centigrade:22   Fahrenheit:71
****Saturday*****
Centigrade:25   Fahrenheit:77
****Sunday****
Centigrade:31   Fahrenheit:87
****Average****
Centigrade:26   Fahrenheit:78
```