

Choices

The example programs so far were all similar in one respect: they all executed in a "straight line". Each statement in the source program executed after the statement on the preceding line. This is convenient if our goal is to learn the basic elements of Java programs, however the demo programs were not very interesting. By providing a capability to "make choices" or to perform simple tasks "repeatedly" or "in a loop", we can devise much more exciting programs. We can begin to solve non-trivial problems, and, more importantly, we can begin to see why the string literal `"Java is fun!"` was used so much in the preceding examples.

We will now study how to alter the straight-line flow of programs by making choices with the `if` statement and the `switch` statement, and by executing a series of statements in a loop using the `while`, `do/while`, and `for` statements. Let's begin with making choices.

Making a choice in a computer program is like confronting a fork in a road. The route taken determines the sights encountered. The choice is something like, "For a scenic trip, turn left; for a fast route, turn right". Such decisions also surface in programming language like Java. Let's begin our examination of programming choices with one of the oldest tools in computer programming: *flowcharts*. Figure 1 shows two flowcharts illustrating (a) a straight-line program and (b) a program that includes a choice.

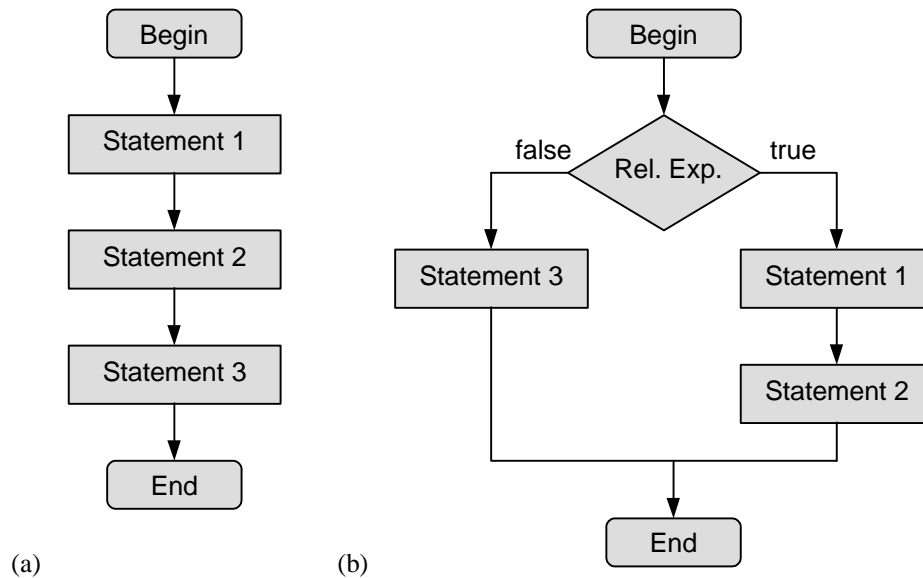


Figure 1. Program flow (a) straight-line execution (b) choice

There are detailed conventions on the design of flowcharts; however, they are not presented in this text. For our purposes, the three basic shapes shown in Figure 1 will suffice. The shapes are connected by arrows showing the flow from one program element to the next. Program termination points are depicted within rounded rectangles using the labels "begin" and "end". These refer to the terminal points for an entire program, a method, or any other subsection of code. Statements are enclosed in rectangles, and decisions are enclosed in diamonds. A decision is always based on a relational expression with `true` or `false` value.

Unfortunately, the picture in Figure 1b is awkward to convert to computer instructions. The problem is that program statements are entered line-by-line, one after the other. At some point,

the two-dimensional layout in Figure 1b must be reworked in a one-dimensional format, as shown in Figure 2.

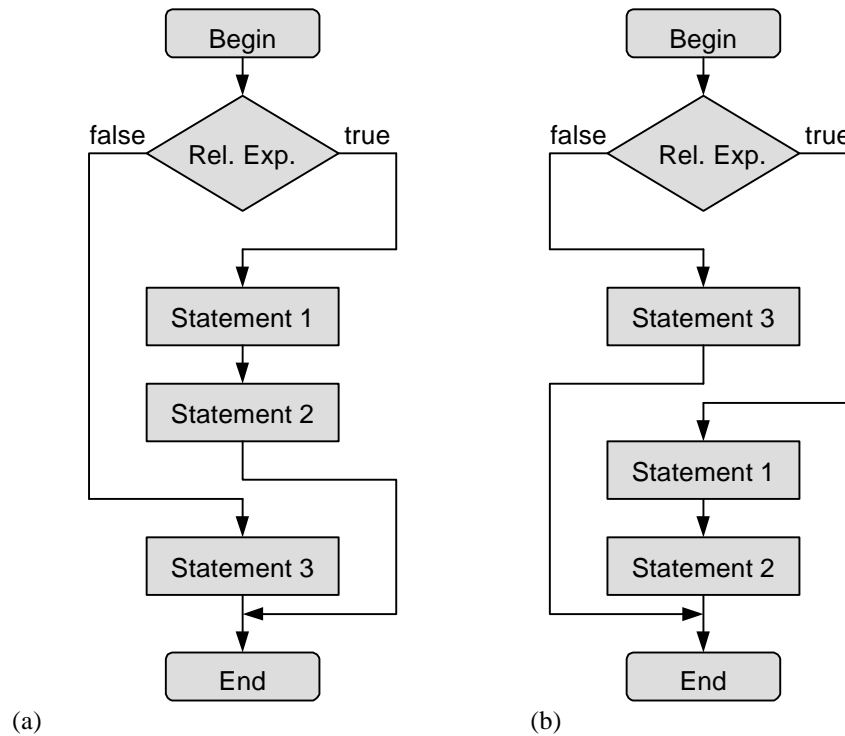


Figure 2. Straight-line flowcharts with choices

Each of part (a) and part (b) in Figure 2 is identical in flow to Figure 1b. However, the pictures are considerably less "pretty". The most useful information in Figure 2 lies in the lines and arrows, rather than in the juxtaposition of the shapes, as in Figure 1b. Figure 1b one more clearly represents a human's mental model of flow with decisions, whereas Figure 2 more clearly represents the line-by-line linear layout of program statements in a source file. The point is this: flowcharts are considerably less useful than they are often made out to be. In this author's experience, programmers rarely solve problems by first constructing a flowchart and then developing the code using the flowchart as a guide. Even if a flowchart is a requirement of an assignment in a programming course, students write and debug the code first, then construct the flowchart later to meet the requirements of the assignment. Is this a bad approach? Perhaps not. Forcing the development process along ordered, isolated activities is usually overstressed and probably wrong. As research in artificial intelligence has discovered, modeling human intelligence is a slippery business. Humans appear to approach the elements of a situation in parallel, simultaneously weighting possible actions and proceeding by intuition.

We will use flowcharts only sparingly from this point onward.

A much more useful aid to the design of programs with non-linear flow is a technique inherited from structure programming languages, such as C or Pascal. The technique is *indentation*. Indentation is, by far, the best way to reveal the flow or possible paths that a program may follow during execution. Not adhering to a consistent style of indentation is an invitation for trouble. The style shown in these notes is consistent. You may adopt it, or adopt some other style; but, be forewarned, as soon as your style of indentation wavers, you will introduce bugs that take far more time to correct than the time to indent your source code. As a guideline, make sure every line of source code has the proper indentation. If you aren't sure how much indentation is needed,

stop! Take a few moments and consider the big picture. How does this statement interact with the statements around it? Don't proceed until you are sure. Okay, so much for the sermon, let's begin our study of program flow by examining Java's `if` statement.

if Statement

In a computer program, we don't usually think about roads and which turn leads to a scenic route. We are more likely to confront situations like this: "If the user typed 'Q', quit the program, otherwise continue". Note the use of the word "if" in the preceding phrase. Not surprisingly, our study of program flow begins with Java's `if` statement.

There are two forms of the `if` statement: `if` and `if/else`. The syntax of the `if/else` form is

```
if (relational_expression)
    statement1;
else
    statement2;
    statement3;
```

where "*relational_expression*" is any term or expression with a boolean value. If the relational expression is `true`, statement #1 executes, otherwise statement #2 executes. After statement #1 or #2 executes, the program continues with statement #3. Any statement can be replaced by a statement block using braces:

```
if (relational_expression)
{
    statement1a;
    statement1b;
    statement1c;
}
else
    statement2;
    statement3;
```

The `else` part is optional:

```
if (relational_expression)
    statement1;
    statement2;
```

With this form, statement #1 executes only if the relational expression is true. Either way, the program proceeds with statement #2. Figure 3 shows the flowcharts for an `if/else` statement and an `if` statement.

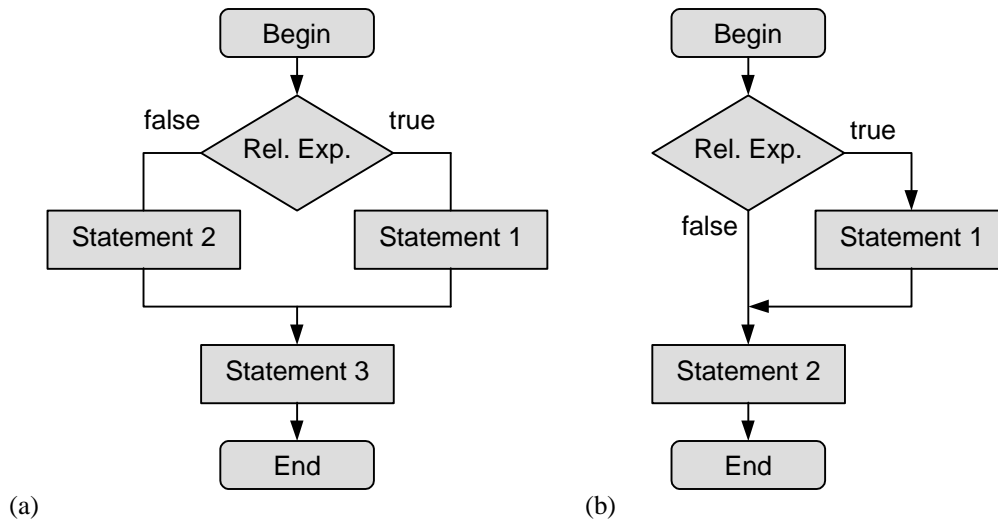


Figure 3. Flowcharts (a) if/else statement (b) if statement

Let's examine the if/else statement in a simple demonstration program. DemoIfElseStatement prompts the user to enter his or her age. The outputs depends on the value entered (see Figure 4).

```

1  import java.io.*;
2
3  public class DemoIfElseStatement
4  {
5      public static void main(String[] args) throws IOException
6      {
7          BufferedReader stdin =
8              new BufferedReader(new InputStreamReader(System.in), 1);
9
10         System.out.println("Welcome to the Java Roadhouse!");
11         System.out.print("Please enter your age: ");
12         int age = Integer.parseInt(stdin.readLine());
13         if (age < 19)
14             System.out.println("Here's a ticket for a free soda");
15         else
16             System.out.println("Here's a ticket for a free beer");
17     }
18 }
  
```

Figure 4. DemoIfElseStatement.java

All the elements in this program were discussed earlier, except for the if/else statement in lines 13-16. Two sample dialogues are shown below. (User input is underlined.)

```
PROMPT>java DemoIfElseStatement
Welcome to the Java Roadhouse!
Please enter your age: 17
Here's a ticket for a free soda
Please enter
```

```
PROMPT>java DemoIfElseStatement
Welcome to the Java Roadhouse!
Please enter your age: 23
Here's a ticket for a free beer
Please enter
```

In line 13, an `if` statement uses the relational expression "`age < 19`" to determine which of two statements to print. The variable `age` is compared with 19 using the `<` operator. The result is a boolean value, `true` if `age` is less than 19, `false` if `age` is 19 or greater.

In many programming situations the `else` part of an `if` statement is not necessary. For example, the following statement

```
if (temperature < 15)
    System.out.println("Don't forget your coat!");
...
```

prints a reminder to bring your coat if the variable `temperature` is less than 15. If `temperature` is 15 or greater, the reminder is not printed. Either way, the program continues with the next statement. Note above that there is no statement that executes only if the `temperature` is 15 or greater.

Nested `if` statements

It is quite common to have `if` statements nested inside other `if` statements to provide multiple alternatives in a programming problem. This is illustrated in Figure 5.

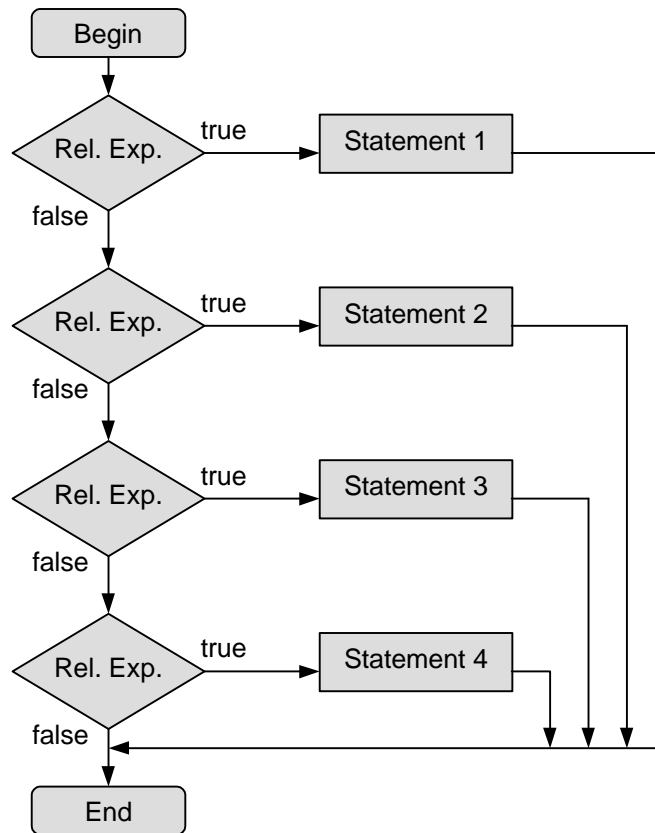


Figure 5. Nested if statements for multiple alternatives

An example is shown in the program `Grades.java` which outputs a letter grade based on a numeric mark.

```

1  import java.io.*;
2
3  public class Grades
4  {
5      public static void main(String[] args) throws IOException
6      {
7          BufferedReader stdin =
8              new BufferedReader(new InputStreamReader(System.in), 1);
9
10         System.out.print("Enter your mark in percent: ");
11
12         double mark = Double.parseDouble(stdin.readLine());
13
14         if (mark >= 80)
15             System.out.println("You got an A");
16         else if (mark >= 70)
17             System.out.println("You got a B");
18         else if (mark >= 60)
19             System.out.println("You got a C");
20         else if (mark >= 50)
21             System.out.println("You got a D");
22         else
23             System.out.println("You got an F");
24     }
25 }

```

Figure 6. Grades . java

A sample dialogue with this program follows:

```

PROMPT>java Grades
Enter your mark in percent: 66.6
You got a C

```

At the point where a relational expression yields true in Grades.java, the letter grade is printed, and no further relational expressions are evaluated. Note that the last else handles the "none of the above" condition.

It can be difficult determining the pairings for the if and else parts of nested if/else statements (particularly if indentation is inconsistent). The rule is this: an else is associated with the nearest preceding if that does not have an else immediately following it.

Figure 7 illustrates nesting if/else statements within each half of a preceding if/else statement.

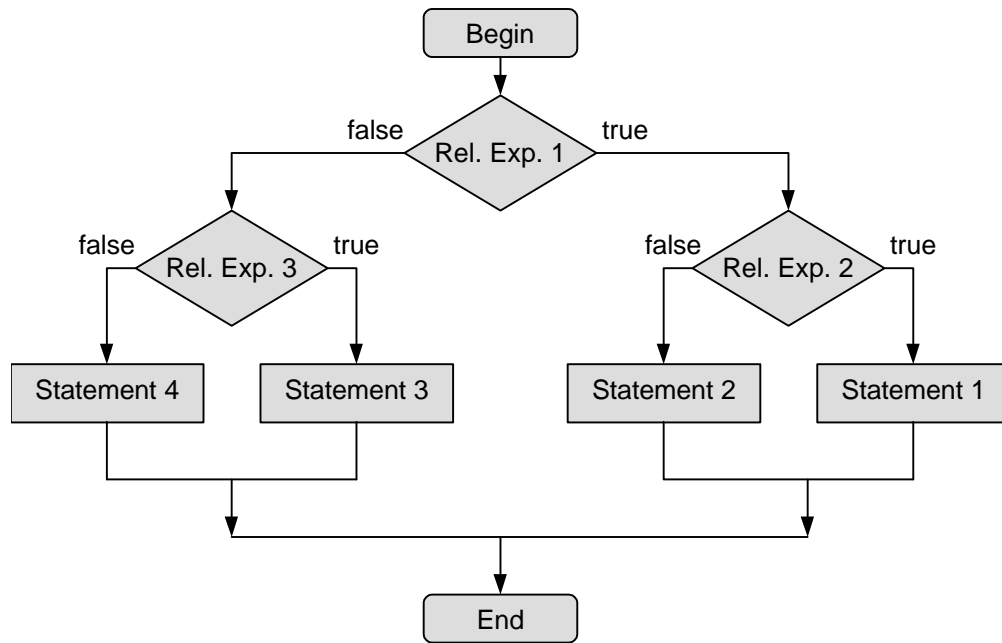


Figure 7. Nesting if/else inside if/else

Two possible implementations are given in Figure 8.

<pre> if (rel_exp1) if (rel_exp2) statement1; else statement2; else if (rel_exp3) statement3; else statement4; </pre> <p>(a)</p>	<pre> if (rel_exp1 && rel_exp2) statement1; else if (rel_exp1 && !rel_exp2) statement2; else if (!rel_exp1 && rel_exp3) statement3; else if (!rel_exp1 && !rel_exp3) statement4; </pre> <p>(b)</p>
--	--

Figure 8. Two implementations for the nested if/else statements in Figure 7.

The code in Figure 8a is clearly easier to follow; however the code in Figure 8b takes fewer lines. The solution in Figure 8b combines relational expressions to more directly arrive at the desired statement. Note the important role of the logical ! (NOT) operator; it effectively captures the "false" paths through the if/else statements.

There are advantages and disadvantages to each approach in Figure 8. In Figure 8a, only two relational expressions are evaluated in reaching the correct statement. In Figure 8b, as many as eight relational expressions may be evaluated. So, in general the approach in Figure 8b is slower.

The approach in Figure 8b also suffers from other "possible" problems. If statement #4 is to be executed, then `rel_exp1` is evaluated four times. Not only is this time consuming, if the expression includes any operations that change the content of a variable in the expression (e.g., `x++`), then the variable will not hold the same value each time the expression is evaluated. There may be a subtle, hidden bug lying in wait! Here's another "possible" problem with the approach in Figure 8b. Due to lazy evaluation (see below), it is not certain how many times `rel_exp2` or

`rel_exp3` are evaluated. It is not possible, therefore, to predict the final value of any variable used in these expressions that changes. Again, a bug may exist that only surfaces under certain initial conditions.

Pay special attention to the effect of lazy evaluation for logical operators. In the first `if` expression in Figure 8b, `rel_exp2` is not evaluated if `rel_exp1` is `false`. (If `rel_exp1` is `false`, the combined expression *must be false*!) This is not a bug; it is simply a behaviour that must be understood by Java programmers. The trick is to make sure no variables used in `rel_exp2` are assigned new values!

Despite the caveats above, programmers still construct `if/else` hierarchies with complex relational expressions. Is this bad? Not necessarily! As noted earlier, humans approach problems and their solutions in a variety of complex and intuitive ways. To stifle such creativity by imposing rigid coding guidelines probably brings forth more problems than those guidelines are worth. The main prerequisite to proceeding creatively in solving problems in Java is a thorough understanding of the constructs of the language.

While we're on the subject of lazy evaluation, let's examine a powerful technique to incorporate safeguards in relational expressions. Suppose `s` references a string of characters. Suppose further that we want to do something special if the string begins with 'Z'. This could be achieved as follows:

```
if (s.substring(0, 1).equals("Z"))
    System.out.println("Begins with Z");
```

If, by chance, `s` references an empty string, we're in trouble because accessing a non-existent index in a string is illegal and will cause a run-time error or exception. This is effectively averted as follows:

```
if (s.length() > 0 && s.substring(0, 1).equals("Z"))
    System.out.println("Begins with Z");
```

If `s` references an empty string, the first relational expression above yields `false`. Since the `&&` (AND) operator connects the two relational expressions, the combined result *must be false*. Therefore, due to lazy evaluation, the second expression is not evaluated and the error is averted.

Selection Operator

The *selection* operator (`? :`) provides a convenient shorthand notation for certain conditional assignments. For example, the following `if/else` statement

```
if (a < b)
    z = a;
else
    z = b;
```

can be coded as

```
z = a < b ? a : b;
```

The variable `z` is assigned the value of `a` (if `a` is less than `b`) or the value of `b` (if `a` is greater than or equal to `b`).

The general syntax for the `? :` operator is

```
rel_exp ? exp1 : exp2
```

This is just an expression which returns the value of *exp1* (if *rel_exp* is true) or *exp2* (if *rel_exp* is false). Expression *rel_exp* is a relational expression, whereas *exp1* and *exp2* are expressions returning any data type or an object. Since the *? :* operator takes three operands, it is called a *ternary* operator.

switch Statement

As with the selection operator, the *switch* statement provides a convenient alternative (but it is not essential). Its main use is to avoid an unsightly series of *if/else* statements when choosing one alternative among a set of alternatives. The syntax for *switch* statement follows:

```
switch (integer_expression)  
{  
    case 0:  
        statement0;  
        break;  
    case 1:  
        statement1;  
        break;  
    case 2:  
        statement2;  
        break;  
    ...  
    default:  
        default_statement;  
}
```

Note that "switch", "case", "break", and "default" are reserved words in Java. If the integer expression equals 0, statement #0 executes, then the *break* statement executes. The *break* statement causes an immediate exit from the *switch*. If the integer expression equals 1, statement #1 executes, and so on. If no case matches the integer expression, the default statement executes. If the *break* statements are omitted, the correct case is still chosen; however, execution proceeds through the other statements until the end of the *switch* or until a *break* statement is eventually reached. This is usually a programming error. The *break* statement is discussed in more detail later.

The program `TaxRate.java` demonstrates the *switch* statement.

```

1  import java.io.*;
2
3  public class TaxRate
4  {
5      public static void main(String[] args) throws IOException
6      {
7          BufferedReader stdin =
8              new BufferedReader(new InputStreamReader(System.in), 1);
9
10         System.out.println("What is your income bracket?");
11         System.out.println("1 = less than 25K");
12         System.out.println("2 = 25K to 50K");
13         System.out.println("3 = more than 50K");
14         System.out.print("Enter: ");
15         int bracket = Integer.parseInt(stdin.readLine());
16         switch (bracket)
17         {
18             case 1:
19                 System.out.println("Pay no taxes");
20                 break;
21             case 2:
22                 System.out.println("Pay 20% taxes");
23                 break;
24             case 3:
25                 System.out.println("Pay 30% taxes");
26                 break;
27             default:
28                 System.out.println("Error: bad input");
29         }
30     }
31 }

```

Figure 9. TaxRate.java

A sample dialogue follows:

```

PROMPT>java TaxRate
What is your income bracket?
1. less than 25K
2. 25K to 50K
3. more than 50K
Enter: 2
Pay 20% taxes

```

This program implements a simple menu. The user is prompted to enter 1, 2, or 3 to select an income bracket. The value is inputted and converted to an `int` in line 15, then passed as an argument to the `switch` statement in line 16. In the dialogue above, the user entered "2", so the message printed was "Pay 20% taxes".

The statements in lines 16-29 could be coded using a series of `if/else` statements as follows:

```

if (bracket == 1)
    System.out.println("Pay no taxes");
else if (bracket == 2)
    System.out.println("Pay 20% taxes");
else if (bracket == 3)
    System.out.println("Pay 30% taxes");
else
    System.out.println("Error: bad input");

```

The arrangement above takes fewer lines, but it's also less clear. The difference is more dramatic is the statements are replaced by statement blocks. The choice of using a `switch` statement or a series of `if/else` statements is a personal preference, and we'll not adhere strictly to either style in these notes.

The `switch` statement only works with integer arguments. This is a limitation in some situations. Since an `if` statement uses a relational expression, it is more flexible. However, with some "preprocessing" a `switch` can often still be used. For example, assume a program inputs words and processes them differently depending on their length. If `word` is a `String` object variable referencing the inputted word, then the processing could proceed as follows using `if/else` statements:

```

if (word.length() == 1)
    ...
else if (word.length() == 2)
    ...
else if (word.length() == 3)
    ...
else
    ...

```

Each ellipsis (`...`) would be replaced by a statement or statement block, as appropriate. The final `else` handles the "none of the above" situation of `word` referencing an empty string. (An empty string has length zero.) The same effect is achieved using a `switch` statement as follows:

```

int len = word.length();
switch (len)
{
    case 1:    // length is 1
        ...
        break;
    case 2:    // length is 2
        ...
        break;
    case 3:    // length is 3
        ...
        break;
    default:   // empty string (length is 0)
        ...
}

```

Besides the more appealing look, the arrangement above has one other benefit. The expression "`word.length() == n`" in the preceding example is re-evaluated for each `if`-test. This is time consuming since it requires a call to a class method. On the other hand the integer comparison in the `switch` statement is extremely fast. The speed difference may not appear in small programs, but it may be a concern in some situations.

Let's reconsider this example with a slightly different goal. Suppose the words are processed separately depending on the first letter in the word. Can a `switch` statement be used? Yes, but a little ingenuity is required. Here's the approach. First, we initialize an `int` variable `code` either with `-1` if `word` references an empty string, or with the lexical difference from `'a'` of the first letter in `word`:

```
int code;
if (word.length() == 0)
    code = -1;
else
    code = word.substring(0, 1).toLowerCase().compareTo("a");
```

So, if the word is "able", `code` is 0, if the word is "baker", `code` is 1, if the word is "charlie", `code` is 2, and so on. It is important to check for an empty string, as above. The expression `word.substring(0, 1)` generates a run-time error (the program crashes!) if `word` references an empty string. The arrangement above effectively guards against this. With this preparation, the words are processed using a `switch` statement as follows:

```
switch (code)
{
    case 0:        // word begins with 'a' or 'A'
        ...
        break;
    case 1:        // word begins with 'b' or 'B'
        ...
        break;
    case 2:        // word begins with 'c' or 'C'
        ...
        break;
    ...
    default:      // empty string or doesn't begin with a letter
        ...
}
```