Appendix C Control Structures

Control Structures

Within the body of a method, there are several forms that can be used to decide which commands are to be performed and in what order to perform them.

Warning:

All control structures have 'heading lines' which should not have semi-colons at the end. Don't put semi-colons at the end of the heading lines! Doing so causes errors, often run time errors that are notoriously hard to find and correct.

The *if* structure

```
if (true or false expression)
    command;
```

or

```
if (true or false expression)
    {
    command;
    command;
    command;
    }
```

any number of commands

The *if* structure can be extended to give two alternative sets of commands, one set will be done:

```
if (true or false expression)
    command;
else
    command;
```

A single command can be replaced by a group of commands (as in the example above).

The *if* structure can also be extended indefinitely in the following way. One set of commands will be done:

```
if (true or false expression)
    command;
else if (true or false expression)
    command;
else if (true or false expression)
    command;
else if (true or false expression)
    command;
else
    command;
```

A single command can be replaced by a group of commands.

The last *else* and the last command can be omitted if there aren't any commands to do when all of the expressions are false.

The *while* structure

The form of the *while* structure is just like *if* . There is no *else* option, however.

```
while (true or false expression)
    command;
```

or

```
while (true or false expression)
    {
    command;
    command;
    command;
    }
```

any number of commands

While can be described as a "stubborn *if*". It works like *if*; but, in addition, if the expression is true, then after doing the command (or commands), the expression is checked again. If the expression remains true, the commands are repeated. This is done over and over until, after the commands have been repeated (for perhaps the umpteenth time), the expression has become false.

The example below draws several horizontal lines (each one 10 dots below the other).

```
int row = 20;
while (row <= 100)
    {
    g.drawLine(10, row, 100, row);
    row = row + 10;
    }
```

WARNINGS:

- Don't put a semi-colon at the end of the line starting with *while*! If you do the loop will get 'stuck' comparing *row* and *100* over and over. This misbehavior is very common, and has cost computer programmers very many hours of frustration.
- If the expression never becomes false, the computer will 'get stuck' performing the commands over and over, and the program will appear to 'freeze'. This is a very common error in computer programs.

The *for* structure

The *for* structure is a shorthand way of writing a *while* structure. It is almost exactly equivalent to the *while* structure. There is one technical difference which is discussed after the examples.

```
for (initialization; true or false expression; incrementation)
   command;
```

or

```
for (initialization; true or false expression; incrementation)
   {
   command;
   command;                          any number of commands
   command;
   }
```

The following structures are almost exactly equivalent:

1.
```
int row = 20;
while (row <= 100)
   {
   g.drawLine(10, row, 100, row);
   row = row + 10;
   }
```

2.
```
int row;
for (row = 20; row <= 100; row = row + 10)
   g.drawLine(10, row, 100, row);
```

3.
```
for (int row = 20; row <= 100; row = row + 10)
   g.drawLine(10, row, 100, row);
```

The initialization part of a *for* structure can be left blank as can the incrementation part. Thus, the following example is also equivalent (although hardly reasonable).

4.
```
int row = 20;
for (; row <= 100;)
   {
   g.drawLine(10, row, 100, row);
   row = row + 10;
   }
```

All of the examples in this section draw the same lines on the screen. The only difference is that in example #3 (the variable *row* is declared in the initialization section of the *for* structure), the variable *row* is not available for use in commands that might follow the for structure. The variable that controls a loop generally should not be used outside of the loop, thus example #3 is usually appropriate.

WARNING:

- Don't put a semi-colon at the end of the line starting with *for*! If you, version 3 will report a syntax error — the others will draw a single line with row equal to 100. This misbehavior is very common, and has cost computer programmers very many hours of frustration.

The *do…while* structure

The *do…while* structure works like the *while* structure except that commands are done the first time without checking the true or false expression. After the first time, *do…while* and *while* act in the same way.

```
do
    command;
while (true or false expression);
```

or

```
do
    {
    command;
    command;      any number of commands
    command;
    }
while (true or false expression);
```

The *switch* structure

The *switch* structure is a shorthand form of *if – else if – else…* that can be used only in certain restricted circumstances. The principle reason for using the *switch* structure is convenience. It is true, however, that when used correctly, the *switch* structure may produce faster programs than the equivalent *if – else* structure.

```
switch (integer type variable)
    {
    case integer type constant:
    case integer type constant:          any number of cases
        command;                         any number of commands
        break;
    case integer type constant:          any number of cases
        command;                         any number of commands
        break;
    case integer type constant:          any number of cases
        command;                         any number of commands
        break;
    default:
        command;                         any number of commands (including just a semi-colon)
    }
```

The integer type variable should be a variable of type *byte*, *short*, *int*, *long* or *char*.

The case constants are values that the variable might attain. Each case constant must have a different value.

When the *switch* structure is translated, an array may be produced with subscripts ranging from the smallest case constant to the largest. If the difference between the smallest and largest constants is very great, a very large array may be produced. For this reason, do not write *switch* statements with widely separated constant values; use *if – else* instead.

WARNING:

DON'T FORGET the BREAK command after each group of commands. Leaving out *break* commands will not create syntax errors, but the program will not run properly.