# 3     **Good Programming Practice**

## 1     Terminology

### Basic elements

You should understand the basic terminology used to describe programs. The most important basic terms are:

> *variable*     *type*     *statement*     *declaration*     *expression*

Examine the following code, for example.

```
int n, k;
System.out.print("Enter a number:");
n = Console.readInt();
k = 0;
while (k*k<n) {
    k = k+1;
}
```

- It employs two *variables*, n and k.
- It has one *declaration statement* (int n,k;) which contains two *declarations* (a declaration of n and a declaration of k).
- It employs three *types*: int (for example, the value 0), String ("Enter a number:"), and boolean (k*k<n).
- It has five *statements* (in addition to the declaration statements). These are the print statement, the two initial assignments (to n and k, respectively), the statement which increments k, and the while-statement (which contains a statement within it).
- It has five (maximal) *expressions*. These are 0 (of type int), k*k<n (of type boolean), "Enter a number" (of type String), k+1 (of type int), and Console.readInt() (of type int). (By maximal expression we mean an expression that is not part of a larger expression. For example, k*k is also an expression but it occurs as a sub-expression of k*k<n.)

It may not appear at first sight that 0 is an expression, but indeed it is – an expression is just a piece of text which describes a value, whether or not it contains operators that must be evaluated. For the same reason, "Enter a number" is an expression, this time of type String. String expressions may contain operators (as in "I earn " + pay + " pounds per annum."), but there are none is the above code.

*Important!* Every variable and every expression has a type. You cannot write programs without knowing the type of every variable and every expression you use.

There is an important distinguish between statements and expressions. On the one hand, statements change the world, for example by making text appear on a screen (as in System.out.print(3.14159);), or by changing the contents of a variable in the machines memory (as in x = (x+y)/2;). On the other hand, expressions inspect the world to compute an answer, for example by examining (without changing) the values of variables, or by sensing whether input has been terminated.

*Important!* Understanding the distinction between statements and expressions is crucial to writing good methods, as we shall see.

## Side-effects

Console.readInt() is an unusual expression because it not only yields a value (the value keyed in by the user), but it has a *side-effect*. A side-effect is a change in the state of the machine caused by the evaluation of an expression. Here the side-effect is a change in the appearance of the screen: after Console.readInt() is evaluated, the number that was input will be visible on the screen and the cursor will have moved. Expressions with side effects are rare (data input as here will be just about the only case we meet).

## 2    Program presentation

Fellow programmers must be able to understand your programs, and so good textual presentation is extremely important. The most important aspects of presentation are choice of names, indentation, and comments.

## Naming

Variables, classes, methods all have names. You must choose appropriately suggestive names, but do not make names overly elaborate. It is usual for the first letter of class names to be in upper case; other names to begin in lower case (constants sometimes excepted). If you make a

name by running some words together, capitalise each word after the first, or separate the component words with underscores, as in bankAccount or bank_account.

Variables which are declared and used over a small textual area, however, can have short, even one-letter, names provided there are not too many of them in the one place. It is also usual to use one- or two-letter names for simple 'counting variables' (such as a variable that records the number of integers that have been read from the keyboard), or array indexing.

*Very important*! When one or two-letter names are used for variables, it is common to reserve names beginning with i, j, k, l, m, and n for variables of type integer, and names beginning with x, y, and z for variables of type real. For example, it is very bad style to use, say, i for the name of a string variable.
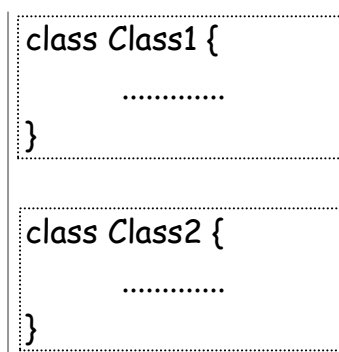
Constants (that is names of values, typically introduced using the keyword final, as in `final double PI = 3.14159`) are often written in all upper case. Either follow this convention, or use some other convention that makes constants clearly recognisable in your program.

## Indentation and spacing

A program has a textual structure as described below.

> *You must make the textual structure of programs visible by good layout.*

A program is a collection of classes. Each class must be clearly identifiable by the eye, typically by positioning its opening and closing chain brackets as shown below, and by separating the classes with a blank line or two. (The lines and boxes below are not drawn in practice, of course. They are there to indicate what the eye should easily pick out when we see the text.)

```
class Class1 {
        .............
}


class Class2 {
        ............
}
```

A class is a collection of variables and methods. The variables should be separated from the methods, and each method should have its own allocated space, separated from other components by a blank line. All the variables and methods should be indented and aligned to

make it textually evident that they belong to the one class, as shown below. Each method must be clearly delineated by laying out its opening an closing chain brackets as indicated:

```
class MyClass {
        double x;
        int y;

        static int distance() {
           ............
        }

        public static void main(String[] args) {
           ...........
        }
}
```

Each method is composed of declarations and statements. The major constituent parts of complex statements (such as loops and if statements) should be made evident by positioning brackets the opening and closing chain brackets as indicated, and by indenting:

```
void display(int count) {
        int k, val;
        k = 0;
        while (k != count) {
                val = Console.readInt();
                if (val<0) {
                        System.out.println(-val);
                        k++;
                }
                else {
                        System.out.println(val);
                }
        }
}
```

If a branch of an if-statement or the body of a loop consists of just one statement with chain brackets omitted, you should nevertheless indent:

```
if (val<0)
        System.out.println(-val);
```

```
else
        System.out.println(val);
```

Alternatively, the statements need not be given their own lines, as in:

```
if (val<0) System.out.println(-val);
else System.out.println(val);
```

Remember that indentation is done only to aid the human reader: it does not affect the behaviour of the program. For example, the following code does not print out the first n integers as the indentation would seem to suggest:

```
k = 0;
while (k<n)
        System.out.println(k);
        k++;
```

The program's behaviour is identical to that of

```
k = 0;
while (k<n)
        System.out.println(k);
k++;
```

 (and this is the proper indentation). What was probably intended was:

```
k = 0;
while (k<n) {
        System.out.println(k);
        k++;
}
```

Proper indentation and spacing is not an option: you must lay out every program to reveal its structure. Some programmers prefer to write opening chain brackets on a separate line, as indicated below, and that style is perfectly acceptable.

```
class MyClass
{
      static int distance()
      {
            .............
      }
      ...........
}
```

## Commenting

You must explain with comments any program code that is not obvious (but do not comment on code that is easily understood on its own). Add comments to explain the role of each major block of code. In particular each non-trivial method should begin with a comment explaining what it does and how it uses its parameters. The purpose of all variables should be made evident, and this usually requires a comment (an exception would  a simple counting variable used over a small region of text whose purpose is immediately obvious). We will return to these points from time to time.

Learning good presentation takes practice: note the presentation in every program you read in books, and imitate. Note, however, that in textbooks much of the explanation of programs is in the body of the text, and so programs are not always commented as they would be in practice.

## 3    Program tracing

*A program is designed and implemented by thinking, not by trial-and-error.*

Programs are designed and implemented by thinking very carefully with complete attention to detail, no matter how minor it may seem. We do not draft something roughly,  and then try to beat it into life at the keyboard. There are many ways in which to think about the behavior of a program. A simple but effective one is called *tracing*. Tracing a program means mimicing its execution on paper for a particular input. We do this by keeping a running account of the value in each variable and the appearance of the screen, carefully recording each change as the statements of the program are executed. It is important to blindly follow the code, doing exactly what it says – the idea is to pretend you are a dumb computer doing exactly what the code says without applying any intelligence. We show an example.

The program below reads a natural number (the natural numbers are 0, 1, 2, 3, 4 etc.) and determines whether it's a perfect square or not (the perfect squares are 0, 1, 4, 9, 16, etc.). An example of input/output is (input in italics):

```
Enter a natural number: 9
That's a perfect square.
```

And another one is:

```
Enter a natural number: 7
That's not a perfect square.
```

We show a program trace for input 7. Tracing is a dynamic process, not easily reproducible on a static page, and so the following looks clumsier than it is in practice. We have labeled each statement and boolean expression in the program for ease of reference. The trace is recorded in the table below. The tables records the values in all variables (here n and k), and the appearance of the screen (with _ representing the cursor). We also have entries for the boolean conditions in loops and if-statements. Each line of the table represents the execution of a statement or the evaluation of a boolean expression.

(The actions carried out by the program for the input in question is indicated in the leftmost column.) When the program trace is complete, we check that the output is as expected for the input we supplied. In this case, the screen reports that 7 is not a perfect square, as we would expect.

```
      class PerfectSquare {
            public static void main(String[] args) {
                  int n, k;
1                 System.out.print("Enter a natural number: ");
2                 n = Console.readInt();
3                 k = 0;
4                 while (k*k<n) {
5                       k++;
                  }
                  // Now k is the smallest (natural) number whose square is >= n
6                 if (k*k==n)
7                       System.out.print("That's a perfect square.");
                  else
8                       System.out.print(" That's not a perfect square.");
```

```
                }
        }
```

| after | n | k | k*k<n | k*k==n | Screen |
|---|---|---|---|---|---|
| 1 | | | | | Enter a natural number: _ |
| 2 | 7 | | | | Enter a natural number: 7 <br><br> _ |
| 3 | | 0 | | | |
| 4 | | | True (0*0<7) | | |
| 5 | | 1 | | | |
| 4 | | | True (1*1<7) | | |
| 5 | | 2 | | | |
| 4 | | | True (2*2<7) | | |
| 5 | | 3 | | | |
| 4 | | | False (3*3<7) | | |
| 6 | | | | False  (3*3=7) | |
| 8 | | | | | Enter a natural number: 7 <br> That's not a perfect square._ |

## 4    Program testing

Every program must be thoroughly tested. We do this by repeatedly executing the program for both typical and untypical inputs, and confirming that each in each case the output is as we expect. It is important to test the program for a wide range of representative inputs; we give some examples below:.

*Program LargerOfTwo*
1. Larger first: 4  3
2. Larger last: 3  4
3. Equal:     4  4
4. Some negatives: -4  5

*Program CharCount*
1. All letters:  abcdcde
2. All digits:  4323456
3. Letters followed by digits:  asght332135
4. Digits followed by letters: 34345dfaad

5. Arbitrary input: 1ytr47yy!,:h67  &97
6. Empty input:

*Program to determine if an integer read in is a perfect square:*
1. A perfect square: 16
2. Not a perfect square: 13
3. Zero: 0
4. A negative number: -16 (the program given earlier will not handle this case)

*Program CountLines*
1. A single line with text
2. A single empty line
3. A sequence of several lines, some of which are empty.
4. No lines (i.e. the only input supplied is the end-of-input indicator).

*Program which reads a line of integers and prints it in ascending order:*
1. A random sequence (with repetitions): 3 6 5 –7 3 –2 –2 5
2. An ascending sequence: –3 –2 –2 0 0 1 4 4 7
3. A descending sequence: 7 4 4 1 0 0 –2 –2 –3
4. A level sequence: 4 4 4 4 4 4
5. A sequence with one value: 4
6. An empty sequence:

# 5    Debugging

If a program under test does not produce the expected output, then you have to find the source of the problem and fix it. This is called "debugging".

**Programs are debugged by thinking, not by trial-and-error.**

Examine the output and try to reason out the kind of erroneous behaviour that could have given rise to it. It will probably not be immediately obvious, and you will have to work at it. Here are some tips.

1. Print out the program text, and study it very carefully. Be a thinking detective.

2. If the program aborted, read the error message that is printed – it often reports quite accurately what went wrong. For example, it might say that a "zero divide" error occurred, meaning that you attempted to divide by zero. If you do not understand the message, don't

ignore it but look it up in a book or ask someone. Many Java programs fail because of "null pointer exception" – if you don't know what this means, find out.

3.  Locate the region of text where the error most likely lies, and focus your attention on this. Error messages will often tell you the line number at which the program aborted. This is a good place to start looking in your program, but bear in mind that the true source of the error could have occurred elsewhere but only showed up at the line indicated. For example, a variable could have been given a wrong initial value, but the problem only shows up when the variable is first used. You may need to insert print statements to help you locate the region where the error occurs; see below.

4.  Try to reproduce the error with as simple an input as possible. For example, if the program is supposed to sort a list of numbers, then try to reproduce the error with a list of just two numbers. Make sure you can reproduce the error every time with this data. Then you will have a good test for any putative repair.

5.  Trace the program on paper for the simplest input that produces the error.

6.  If the program does not terminate, then it is almost certainly in an infinite loop. Examine the loops to see that you haven't forgotten to increment a counter variable in all circumstances. If your program has more than one loop, the output should give you a strong clue as to which loop is cycling. If you are still not sure, insert lots of print statements in the program to find out how far the program progresses. Each added print statement need only print a letter on a line (each prints a different letter, of course). Run the program again and you will immediately able to tell from the output how far the program has progressed.

7.  Add lots of print statements to print out the values of variables as the program progresses. Study the output – the changing values of the variables will give you a strong indication of what may be going wrong.

8.  Think of conditions you expect to be true as various points in the program, and insert print statements to confirm this. Such conditions are called "assertions". For example, if at a particular point, you expect integer variable m, say, to be larger than n squared, then insert `print("m>n*n: ", m>n*n)` at that point and run the program again to confirm your expectations. Alternatively, encode the assertion using an if-statement which prints a message only if the assertion is false.

9.  When you believe you have traced the source of the error and repaired it, test the program on the input data that you exhibited the error originally.

10. Don't proceed by trial-and-error, making haphazard changes that you haven't though through. Thinking systematically is the only way to proceed in confidence.