1 Review of Basic Java

1 Background

The course is a follow-on to your first programming course in Java. You are expected to be familiar with the elements of Java listed below. However, we will do some revision, especially of methods and parameters.

Basic
print() and println(). String constants and string concatenation (+). Displaying strings, integers, doubles, and booleans.
Console input of integers, doubles, characters, strings, ...
main(). Form of programs.

Elementary data types
Integers. + - * / %
Booleans. & && | || ! < == <= > >= !=. Very elementary boolean algebra.
Doubles. + - * / sqrt() round() min() max() abs() etc.
Characters. Conversion to and from integers. isDigit() isLetter() isLetterOrDigit() isLowerCase() isUpperCase() toLowerCase() toUpperCase()

Elementary type conversion (we will take a refresher look at this).

Variables
Declaring and initialising variables. Defining constants. Scope and lifetimes of declarations. Assignments of elementary types. Short forms assignments (as in n++).

Control statements
if-statement.
while- and for-statement etc.
Writing complex boolean expressions in if-, while-, and for-statements.

Static methods

Defining and invoking static methods. Parameters of elementary type. Return types; void. Using static methods to give structure to programs.

Strings

String constants. length(), String.valueOf(). parseInt() parseDouble() etc. equals() compareTo() charAt() startsWith() endsWith() indexOF() substring() trim() toUpperCase() toLowerCase() equalsIgnoreCase() (we will take a refresher look at these later)

Arrays

One-dimensional arrays of elementary types. Array initialisation. Array assignment (as objects!). Arrays of strings. Command-line arguments (to be refreshed).

Practice

Proficiency in using all the above elements in programs. Compiling, testing, and debugging. Design, construction, and testing of programs up to two or three classes or so.

2 Revision programs

Below are some elementary programs to refresh your understanding. You should be able to understand them fully.

Example: Larger of two integers

The following program reads two integers and calculates the larger of the two. An example of input/output is (input is shown in italics):

```
Enter two integers:
-17
23
23 is the larger.
```

```
class LargerOfTwo {

    public static void main(String[] args) {
        int m, n, larger;
        System.out.println("Enter two integers: ");
        m = Console.readInt();  n = Console.readInt();
        if (m>n) {
            larger = m;
```

```
                }
                else {
                        larger = n;
                }
                System.out.println(larger + " is the larger.");
        }

}
```

Note that when chain brackets enclose just a single statement they can be omitted. For example, the if-statement in the preceding program could be equally well written as

```
if (m>n) larger = m;
else larger = n;
```

Example: Counting letters and digits

The following program reads a line of text and counts the number of letters and the number of digits. A typical execution looks like (input is shown in italics):

```
Enter a line of text...
This is 2001.
Number of letters 6
Number of digits 4
```

The program uses method readChar() from class Console; it reads a single character from the keyboard.

```
class CharCount {
        public static void main (String argv[]) {
                int numLetters, numDigits;
                char c;
                System.out.println("Enter a line of text...");
                numLetters = 0; numDigits = 0;
                c = Console.readChar();
                while (c != '\n') {
                        if (c>='0'&& c<='9')  numDigits++;
                        else if ((c>='A' && c<='Z') || (c>='a' && c<='z'))
                                        numLetters++;
                        c = Console.readChar();
```

```
        }
        System.out.println("Number of letters: " + numLetters);
        System.out.println("Number of digits: " + numDigits);
    }
```

Note that the end-of-line character is denoted by `'\n'`. When you wish to read an input until a certain value is entered (in this example, an end-of-line character), you must use a while-loop rather than a for-loop, and you must input both before the loop and at the end of the loop body (seek out for the two instances of `c = Console.readChar()` above). You should understand how &&'s and ||'s are used to encode complex conditions: above, `c>='A' && c<='Z'` encodes "c is an upper case letter", and `(c>='A'&& c<='Z') || (c>='a'&& c<='z')` encodes "c is either an upper case or a lower case letter". You should also understand that if no boolean condition in an if-statement evaluates to True, then the if-statement has no effect. For example, if the value read into c is a punctuation mark (such as a comma or period), then no incrementing of numLetters or numDigits takes place.

3 Text input: Console class

Because Java only supplies primitive methods for inputting text from the keyboard, you are provided with a class called Console which contains useful methods for keyboard input. The following methods read, respectively, an integer, a double, a boolean, and a word (not containing any blanks):

Console.readInt() Console.readDouble() Console.readBoolean()
Console.readToken()

The items to be input may be freely laid out on one or more lines, just as long as the items on each line are separated by one or more spaces.

The following method reads a single character:

Console.readChar()

The character returned could be an end-of-line character (denoted in programs by '\n').

The following method reads a string consisting of all the characters remaining in the current line (the end-of-line-character is discarded):

Console.readString()

Console input is line-buffered. This means that the data entered by the user is passed to the running program only at each press of the return key. For example, if you type three items on a line and then notice a mistake in the second item, say, you can backspace to the error and correct it, and then re-enter the rest of the line. Only when you press the return key is the data irrevocably transmitted to the program.

The following method yields information about the state of the input, without reading anything:

```
Console.endOfFile()
```

It yields True if there is no more input, and otherwise yields False. In DOS, the end of keyboard input is signalled by typing control-Z on a line by itself. As an example, the following program counts the number of lines the user enters:

```
class CountLines {
        public static void main(String[] args) {
                String s;
                int numLines = 0;
                while (!Console.endOfFile()) {
                        s = Console.readString();
                        numLines++;
                }
                System.out.println(numLines + " lines");
        }
}
```

As an alternative way of handling the end of input, both readToken() and readString() return the special value *null* if no input remains. For example, the following is an alternative way to count lines entered at the keyboard:

```
class CountLines {
        public static void main(String[] args) {
                String s;
                int numLines = 0;
                s = Console.readString();  // note preliminary readString() ...
                while (s!=null) {
                        numLines++;
```

```
                s = Console.readString(); // and readString() again
            }
            System.out.println(numLines + " lines");
        }
    }
```

endOfFile() is the more general technique for detecting end of input because only readToken() and readString() return the special value *null* if no input remain (readInt(), readChar() etc *never* return null).

Here is a larger example. It reads a sequence of lines from the keyboard. Each line contains a student mark followed by the name of the student. The program displays the name of the best student (i.e. the one with the highest mark). An example of input is:

```
57 Michael Murphy
89 Patrick James McMahon
78 Jenny Smith
```

This will give rise to the output

```
The best student is Patrick James McMahon who scored 89 marks.
```

The program is

```
class BestStudent{
    public static void main(String[] args) {
        int bestMark = -1; // Best mark seen so far (-1 for clean start)
        String bestStudent = ""; // Name of best student so far.
        //Read student data
        while (! Console.endOfFile()) {
            int m = Console.readInt();  String s = Console.readString();
            // Is this student the best so far?
            if (m>bestMark) {  // Yes, so note the details
                bestMark = m;  bestStudent = s;
            }
        }
        // Print result
        System.out.println("The best student is " + bestStudent +
                                " who scored " + bestMark + " marks.");
```

```
    }
}
```

4 Console class: additional details

In class Console, the methods readToken(), readInt(), readBoolean(), and readDouble() are based on the notion of “tokens”. A token is a maximal sequence of characters other than white space (white space consists of spaces, tabs, and end-of-line characters). For example, the tokens in the following input (where - represents a press of the space bar and ® a press of the RETURN key)

------23----true-X---Java123---the--rest---®
--3.14®

are “23”, “true”, “X”, “Java123”, “the”, “rest”, and “3.14”. readToken() reads and returns the next token from the keyboard (note that the final delimiting white space character is read but discarded). readToken() always returns a string, even it consists entirely of digits, say. readInt() reads the next token, converts it to an integer, and returns that integer value; it is an error if the token does not represent an integer (i.e. if it is not a sequence of digits, optionally preceded by a minus sign). readBoolean() and readDouble() behave analogously.

readChar() reads and returns the next character in the input (which could be a space, tab, or end-of-line character). Executing a sequence of readChar()’s will yield each character in the input, including each end-of-line character. The end-of-line character is always returned as '\n', even in systems which mark the end of line with a carriage return and a linefeed in succession.

readString() reads the remainder of the text on the current line; the delimiting end-of-line character does not appear at the end of the string that is returned (it is discarded). Usually readString() is invoked to get a line of input, but you may also invoke it if you want the execution of your program to be delayed until the user has pressed the return key.

As an example, suppose a program executes the statements

```
i=Console.readInt(); b=Console.readBoolean();
c=Console.readChar(); t=Console.readToken();
s=Console.readString(); d=Console.readDouble();
```

where the user types

------23----true-X---Java123---the--rest---®
--3.14®

Then the variables would acquire values as follows:

i: 23 b: true c:'X' t:"Java123" s:"--the--rest---" d: 3.14

The following method is also provided, although it is not much used

Console.skipLine()

It discards any remaining input supplied on the current line (including the end-of-line character). Console provides two more methods of minor interest:

Console.hasMoreTokens()
Console.available()

Each yields information about the state of the current line, without reading anything. They are primarily intended for use when input is redirected from a file as explained later. Console.hasMoreTokens() yields True if there are more words on the current line, and otherwise yields False. Console.available() yields an integer equal to the number of characters remaining on the current line (the end-of-line accounts for a single character, even in systems which mark the end of line with a carriage return and a linefeed in succession).

An outline of Console is:

```
class Console {

   static int readInt()
   // Consume and return an integer. Trailing delimiter consumed.
   { ... }

   static boolean readBoolean()
   // Consume and return a boolean. Trailing delimiter consumed.
   // Any string other than "true" (case ignored) is treated as false.
   { ... }
```

```
static double readDouble()
// Consume and return a double. Trailing delimiter consumed.
{ ... }

static char readChar()
//Consume and return a character (which may be an end-of-line).
{ ... }

static String readString()
// Consume and return the rest of current line (end-of-line discarded).
// null returned on end of file

{ ... }

static void skipLine()
// Skip any remaining input on this line.
{ ... }

static String readToken()
// Consume and return a token. Trailing delimiter consumed.
// A token is a maximal sequence of non-whitespace characters.
// null returned on end of file

{ ... }

static boolean endOfFile()
// More characters?
// This method is intended for use when keyboard is redirected to file
{ ... }

static boolean hasMoreTokens()
// More words on the current line ?
// This method is intended for use when keyboard is redirected to file
{ ... }

static int available()
// Number of characters available on this line (including end-of-line,
// which counts as one character, i.e. '\n')
// This method is intended for use when keyboard is redirected to file
```

```
  { ... }

}
```

5 Formatted output: printf

`System.out.printf` is similar to `System.out.print`, except that it allows us to control the layout of what is printed. The first argument of `printf` specifies the format of the items to be printed. A typical form is

```
System.out.printf("%5d", 87)
```

which causes “ 87” to be printed (with 3 leading spaces). The format string "`%5d`" indicates that one item will be printed (there’s just one % symbol), it will be a decimal number (indicated by `d`), and it will have a “field length” of 5 (i.e. it will occupy 5 positions) with spaces added as necessary on the left. A string that is padded on the left with spaces is said to be “right-justified”. The spaces can be added on the right rather than the left by inserting a hyphen as follows:

```
System.out.printf("%-5d", 87)
```

-- this will cause to be printed. “87 ”. A string that is padded on the right with spaces is said to be “left-justified”. To print strings use the symbol `s` instead of `d`. For example,

```
System.out.printf("%-8s", "Java")
```

causes “`Java `” to be printed. Multiple items can be printed in one `printf()` statement:

```
System.out.printf("%-8s%5d", "Java", 87)
```

which causes “`Java  87`”.

Characters for printing can also be placed in the format string. For example:

```
System.out.printf("Results: %-8s and %5d", "Java", 87)
```

causes "`Results: Java and 87`" to be printed. This can be used to cause the output to be printed on a line to itself (like println) by including the carriage-return character `"\n", as in`

```
System.out.printf("%-5d\n", 87)
```

The letter `f` is used in format strings to indicate a real number. The number of positions allocated to the fraction part is indicated by writing `.n` after the field length where n stands for the number of positions for the fraction . If the fraction requires more than n positions, it will be rounded off to n decimal places. For example,

```
System.out.printf("%6.2f", 2.7182)
```

causes " `2.72`" to be printed (this 2 leading spaces because the digits and the decimal point occupy 4 positions and the field length is specified as 6).)

6 Redirecting standard input and output

Every execution of a Java program has associated with it a standard input device and a standard output device. By default, the standard input is the keyboard, and the standard output is the screen. The Console class reads from the standard input, and both print() and println() write to the standard output.

It is possible to redirect the standard input and output for each execution of a program. Almost always, the redirection associates the input with a particular text file, and/or the output with another text file. For example, when testing programs it is tedious to have to repeatedly key in the same data each time we run the program. It is easier to type the data into a file, and let the program read from the file rather than the keyboard. Indeed real programs usually process lots of data, prepared in advance and placed in a file. Suppose, for example that we wish to test the program BestStudent above. We prepare a file called, say, students.txt (it doesn't have to have a .txt suffix) containing the desired input data. We can use any text editor (such as NotePad), but not a word processor (such as Word). The file might look like

```
57 Michael Murphy
89 Patrick James McMahon
78 Jenny Smith
```

Then instead of executing the program via java BestStudent, we execute the command

```
java BestStudent < students.txt
```

which will run BestStudent with the standard input redirected to from the keyboard to the file students.txt. As Console reads from the standard input, the program now takes its input from students.txt without any change being needed in the program itself.

You can also cause output to be sent to a file rather than the screen. This is useful if you want subsequently to print the output. The command

```
java BestStudent > output.txt
```

executes BestStudent with the standard output redirected to a file called output.txt. Any file name will do, and the file need not be created in advance. The program will take input from the keyboard, but no output will be displayed on the screen. Instead, file output.txt will be created containing the program's output; it can be inspected after the program has run by using a text editor. The command

```
java BestStudent < students.txt > output.txt
```

will run the program with both input and output being redirected as indicated.

Note on end of input from files: The endOfFile() method determines whether there are any *characters* remaining in the input. Some editors/systems supply an extra end-of-line at the end of the file by default, and so you it is best not to insert a carriage return at the end of the final line if you are using endOfFile(). If you do, endOfFile() may see an extra end-of-line (the one supplied by the system) and will report that there is more data when in fact there isn't. Do not type Control-Z or similar at the end of a text file.

7 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++;
}
```

- 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* (not counting 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 four (maximal) *expressions*. These are 0 (of type int), k*k<n (of type boolean), "Enter a number" (of type String), 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.

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. Generally, changes in the state of the machine are effected by statements (think of an assignment statement which changes the value of some variable, or a print statement which changes the appearance of the screen). The evaluation of an expression typically does not change the state, but merely examines it (it looks at the values

in variables, for example). We generally dislike expressions with side-effects because they can make reasoning about a program difficult. They are rare in well-written programs, input being almost the only place where they occur.

The reals

The “reals” are numbers with decimal points such as 3.14 or 5.0 (although we usually write 0 rather than 0.0). Java provides several types to represent reals, including double and float, of which the most useful is double. In this course, we will always use type double to represent the reals. We will typically say that a certain variable is of type real rather than type double or float.

Global versus local variables.

Variables may be global or local. A variable is said to be *local* to a method if it is declared within the method. Otherwise it is *global*. Parameters are local variables, special only in the manner in which they are used. For example:

```
class MyClass {
        int pay;                    ⟸  global variable

        int display(String s) {
                int k;              ⟸  local variable
                ..........
        }
}
```

8 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, and other names to begin in lower case. 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.

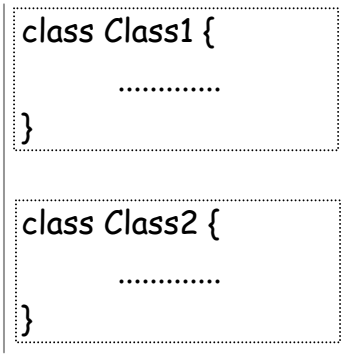
Variables which are declared and used over a small textual area, however, can have short, even one-letter, names provided there are not too many of them. It is also usual to use one- or two-letter names for simple 'counting variables' (such as a variable that records the number of integers that have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

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.)



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.

9 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 _
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 That's not a perfect square._

10 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:

11 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 test 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.

12 Type conversion reviewed

The number 2000, say, can be represented in a program in several ways. If 2000 arises as, say, the maximum number of data items that the program is prepared to process, then 2000 is best represented as an integer. On the other hand, if 2000 arises as the average of a collection of real numbers and it is just accidental that it is a whole number, then we think of 2000 as being the real number 2000.0. Finally, if 2000 is a telephone number then it is appropriate to represent it as the string “2000”. (It would be quite wrong to treat a telephone number as an integer, because many telephone numbers start with 0, and the integer type does not distinguish between, say, 08612345 and 8612345.)

Similar remarks hold for other values. For example, a letter of the alphabet can be represented as a character or as a string of length 1. Usually it is appropriate to represent it as a character of course, but not always.

Occasionally we need to convert representations, i.e. we need to translate a value represented in some type to a corresponding representation in another. For example, we may need to convert the string “2000” to the integer 2000. This is called “type conversion”. The usual mechanism in Java for type conversion is called “type casting”: we write (T) x to convert x to type T. We give some examples of the use of this below, and discuss some other useful type conversion mechanisms.

Converting reals to integers: Math.round()

You can convert a real to an integer explicitly by prefixing it with (int), as in either of the following

```
(int) 3.14
(int)(3.14*2.1).
```

The second pair of brackets in (int)(3.14*2.1) indicates that the type conversion is to apply to the entire expression and not just the first term. The fraction is lost in the conversion. For example, (int) 3.95 is 3. It is usually more appropriate to round off a real before converting it to an integer. For x any real, Math.round(x) yields x rounded off; for example, Math.round(3.95) yields 4.0, and Math.round(3.14) yields 3.0. The best way to convert a real x to an integer is (int) Math.round(x). For example,

(int)Math.round(3.95) = 4.

Converting integers to reals

To convert an integer to a real, prefix it with (double), as in

(double) 4

which yields 4.0. Actually, you don't often have to so this because Java allows you to supply an integer wherever a real would normally be expected; the conversion takes place automatically. For example, it is legal to write 3.14*27 (the result will be a real).

Converting strings to & from integers

If a string s encodes an integer (such as "37" or "-37", say, but not any of "3.14" or "37.0" or "1+3", or "three") then Integer.parseInt(s) yields the encoded value as an int type. For example,

Integer.parseInt("12") = 12.

In many cases, you can supply a number where a string would normally be expected, and java will convert it to the corresponding string automatically. You can exploit this to convert a number to a string: just concatenate it with the empty string. For example,

""+37 = "37"

(note that the + here denotes string concatenation, not addition). This trick works because Java recognises the first argument as a string, treats the + as concatenation, and automatically converts the second argument to a string. Here is one more example:

```
String year = "2001"; // any year
String nextYear = "" + (Integer.parseInt(year)+1); // computes following year
```

The outer brackets in (Integer.parseInt(year)+1) are necessary in order for the inner + to be interpreted as integer addition rather than string concatenation.

Converting strings to reals

Reals are converted to strings as for integers; e.g.

""+3.14 = "3.14".

If a string s encodes a real number (such as "37.3" or "-37.3") then Double.parseDouble(s) yields the encoded value as a double. For example,

Double.parseDouble("3.14") = 3.14.

Converting strings to booleans

The expression ""+b converts boolean value b to a string.

""+b = "3.14".

If string s encodes a boolean (such as "true" or "false") then (new Boolean(s)).booleanValue() yields the encoded value as a boolean. For example, (new Boolean("true")).booleanValue() yields the boolean `true`.