A first problem solving and programming example

Background Integer computation remains important for signal, image and video compression, for encryption, and other applications.

Problem statement Write a program that reads an integer $n > 1$ from standard input and writes all the (possibly repeated) prime factors of n to standard output in nondecreasing order.

Sample inputs

```
84
17
```

Sample outputs

```
84 = 2 . 2 . 3 . 7
17 = 17
```

Question What is the output for inputs `30`? `96`? `777888`?

Question What should the program do if the input is an integer <= 1, or not an integer at all?

A first attempt to understand the problem and design a solution

```
Read integer n ;
Write string "n = ";
for (int c = 2; c <= n; c++) {
    if (c divides n) {
        Write string "c .";
    }
}
Write new line;
```

Sample inputs

```
84
17
```

Resulting outputs (by hand calculation)

```
84 = 2 . 3 . 4 . 6 . 7 . 12 . 14 . 21 . 28 . 42 . 84 .
17 = 17 .
```

This is obviously wrong (why?); we need to think a bit harder!

Understanding the problem better

The previous design seemed to consider each prime factor c repeatedly. So let's try repeatedly *dividing* n by c for each candidate factor c in turn.

84 / 2 = 42
42 / 2 = 21
2 does not divide 21
21 / 3 = 7
3 does not divide 7
4 does not divide 7
5 does not divide 7
6 does not divide 7
7 / 7 = 1
Stop

Hmm, that may lead to a solution...

A second attempt to design a solution

```
Read integer n ;
Write string "n = " ;
c = 2;
while (n > 1) { // eliminate candidate c
    while (c divides n) { // eliminate all occurrences of c
        Write c;
        n = n / c;
        if (n > 1)
            Write ".";
    }
    c = c + 1;
}
Write new line;
```

Hand calculation suggests this is promising... It seems to generate all prime factors (why only prime factors?) in nondecreasing order… Now, how to implement the design in Java?

Implementation issues

1. How to read the integer *n* ?
   - Using TerminalIO?
   - Using standard Java input?
   - From an IntegerField using BreezySwing?
   - From a command line argument? (Violating the specification)

2. How to tell whether or not `c divides n` ?
   - Easy: `n % c == 0` ?

3. What classes and methods should the program contain?
   - Simple program, so single class should suffice.
   - But the program will be clearer and more reusable if we separate reading the input from factorisation, i.e., do the factorisation in a separate method. (It is often good practice to separate i/o from computation. Later, I'll show how to do this for the current problem.)

Version using TerminalIO

```
import TerminalIO.*;

public class Factorise {

    public static void main(String[] args) {
        Factorise app = new Factorise();
        app.doit();
    }

    // Read and print the factors of a positive integer
    public void doit() {
        KeyboardReader in = new KeyboardReader();

        int n = in.readInt();
        factorise(n);
    }

    // (continued next slide)
```


Version using TerminalIO (cont.)

```
    // Print the factors of n to standard output
    public void factorise(int n) {
        ScreenWriter out = new ScreenWriter();
        int c = 2;

        out.print(n + " = ");
        while (n > 1) {
            while (n % c == 0) {
                out.print(c);
                n = n / c;
                if (n > 1)
                    out.print(" . ");
            }
            c = c + 1;
        }
        out.println();
    }

}
```


Version using standard input/output

```
import java.io.*;

public class Factorise {

    public static void main(String[] args)
            throws IOException {
        Factorise app = new Factorise();
        app.doit();
    }

    private void doit() throws IOException {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));

        int n = Integer.parseInt(in.readLine());

        factorise(n);
    }
```

Version using standard input/output (cont.)

```
    private void factorise(int n) {
        int c = 2;

        System.out.print(n + " = ");
        while (n > 1) {
            while (n % c == 0) {
                System.out.print(c);
                n = n / c;
                if (n > 1)
                    System.out.print(" . ");
            }
            c = c + 1;
        }
        System.out.println();
    }

}
```


Notes on standard input/output

1. Remember to import `java.io.*`.
2. Include a "`throws IOException`" clause in every method definition that contains a call to a read method or that calls a method that contains a call to a read method.
3. Declare the `BufferedReader` variable in the innermost scope (method or class) that requires it.
4. Understand the difference between end of file (`line == null`) and an empty line (`line.equals("")`). (See following example.)

Factorising a sequence of positive integers using standard I/O

```
import java.io.*;

public class Factorise {

    public static void main(String[] args)
            throws IOException {
        Factorise app = new Factorise();
        app.doit();
    }

    private void doit() throws IOException {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        String line = in.readLine();
        while (line != null) { // line == null at eof
            int n = Integer.parseInt(line);
            factorise(n);
            line = in.readLine();
        }
    }
```


Factorising a sequence of positive integers using standard I/O (cont.)

```
    public void factorise(int n) {
        /* ... */
    }

}
```

In Windows, indicate end of input (end of file) by **control-Z**, in Unix *(e.g.*, on gucis), indicate end of input by **control-D**.

Note that end of file (`null`, which is not a string) is quite different from the empty string ("").

A problem with all these solutions

Consider the factorisation of 101:

2 does not divide 101
3 does not divide 101
...
10 does not divide 101
11 does not divide 101
12 does not divide 101
...
100 does not divide 101
101 / 101 = 1
Stop.

These solutions are very inefficient. No candidate factor greater than 10 can divide 101, because if it did, some smaller candidate would have already divided 101. We should stop testing immediately if `c*c > n`.

A more efficient factorisation method (for the standard I/O version)

```
public void factorise(int n) {
    System.out.print(n + " = ");
    int c = 2;
    while (n > 1 && c*c <= n) {      // Change!
        while (n % c == 0) {
            System.out.print(c); // String.valueOf(c);
            n = n / c;
            if (n > 1)
                System.out.print(" . ");
        }
        c = c + 1;
    }
    if (n > 1)                          // Change
        System.out.print(n); // String.valueOf(n);
    System.out.println();
}
```


Separating input/output from computation

```
import java.io.*;

public class Factorise {

    public static void main (String[] args)
            throws IOException {
        Factorise app = new Factorise();
        app.doit();
    }

    private void doit() throws IOException {
        BufferedReader in = ...;
        String line = in.readLine();
        while (line != null) { // line == null at eof
            int n = Integer.parseInt(line);
            int[] factors = factorise(n); // list of factors
            print(factors, n);
            line = in.readLine();
        }
    }
```


Separating input/output from computation (cont.)

```
public int[] factorise(int n) {
    // Create a (long) temporary array for factors
    int[] tmp = new int[log2(n)];
    int c = 2, nFactors = 0;
    while (n > 1 && c*c <= n) {
        while (n % c == 0) {
            tmp[nFactors] = c; nFactors++;
            n = n / c;
        }
        c = c + 1;
    }
    if (n > 1) {
        tmp[nFactors] = n; nFactors++;
    }
    // continued...
```

Separating input/output from computation (cont.)

```
    public int[] factorise(int n) {
        // continued from previous slide…

        // Create a shorter results array
        int[] result = new int[nFactors];
        // Copy temporary array to results array
        for (int k = 0; k < result.length; k++)
            result[k] = tmp[k];
        return result;
    }

    // Returns log to the base 2 of n
    // Uses the formula log_2(n) = log_e(n) / log_e(2)
    // (Could have used repeated division by 2)
    public int log2(int n) {
        double log2n = Math.log(n) / Math.log(2);
        return (int) Math.floor(log2n);
    }
```


Separating input/output from computation (cont.)

```
    public void print(int[]factors, int n) {
        System.out.print(n + " = ");
        for (int k = 0; k < factors.length; k++) {
            System.out.print(factors[k]);
            if (k < factors.length-1)
                System.out.print(" . ");
        }
        System.out.println();
    }
}
```


Reading several integers from a single line (see notes on classes and objects)

```
import java.io.*;
import java.util.*;

public class Factorise {

    ...

    private void doit() throws IOException {
        BufferedReader in = ...;
        String line = in.readLine();
        while (line != null) { // line == null at eof
            StringTokenizer t = new StringTokenizer(line);
            while (t.hasMoreTokens()) {  // there is another token
                String s = t.nextToken(); // get next token
                int n = Integer.parseInt(s);
                int[] factors = factorise(n);
                print(factors, n);
            }
            line = in.readLine();
        }
    }
}
```


What has this example illustrated

1. Problem solving can require both experiment and reasoning to find a solution.
2. Problem understanding and design should precede implementation. Designs should be tested by reasoning and hand calculation.
3. Simple patterns can be used repeatedly, for example, to read every line in an input file.
4. It is straightforward to modify programs using TerminalIO to use standard input/output:
   - Import `java.io.*` instead of `TerminalIO.*`
   - `KeyboardReader in = new keyboardReader();`
     is transformed to:
     ```
     BufferedReader in =
         new BufferedReader(
             new InputStreamReader(System.in));
     ```
   - `in.readInt()`
     is transformed to:
     `Integer.parseInt(in.readLine());`
   - Other transformations are similar, and are described later.

What has this example illustrated (cont.)

5. It is also fairly straightforward to transform BreezySwing applications to GUI applications using AWT/Swing!
   – (Described later)

6. Analysis and simple changes can lead to very significant improvements in performance.

7. Programs should be written **without** using static methods (except for method `main`).

8. It is desirable and usually straightforward to separate input/output and computation, here using methods, more generally using separate classes.

What remains to be done?

All these solutions have several faults:

1. They do not test that integer *n* is in fact positive.

2. They may not be properly defined when *n == 1*.

3. They do not test whether errors occur when trying to read (and write), for example if the data entered is not actually an integer.

4. They do not conform to the specified coding conventions. In particular, they do not contain comments. (Justification? Otherwise they would not fit on these tiny slides.)

5. No evidence of program testing has been provided.

6. They are still not efficient enough for actually breaking real encryption systems! :-(