Files: line-by-line or token-by-token

If a file is formatted by line

- Eg, each "item" is described by a sequence of values on a single line

- It is simplest to read every line into a List of lines, and process with a for-each loop

```
List<String> allLines = Files.readAllLines( Path.of(fname) );
for (String line : lines){
    Scanner scan = new Scanner (line);
    ….
```

973 biscuits 27 33 15 4 9
731 cake 3 5 2
189 fruit 54
446 beans

Once upon a time there was a chicken who lived on a little farm in a tiny village, away out in the far country, beyond the

If a file is a sequence of tokens, and the lines don't mean anything

- Eg, long sequence of words, with arbitrary line breaks.
- Eg, long sequence of numbers

- It is simplest to put a single Scanner around the whole file:

```
Scanner scan =  new Scanner( Path.of(fname) );
 …  scan.next() ….  scan.nextDouble()….
```

Summing all the numbers in a file

```
/** Return the sum of all the numbers in a file, ignoring the non-numbers */
public void addNumbers(String fname){
    try {
        Scanner scan = new Scanner( Path.of(fname) );
        double total = 0;
        while (scan.hasNext() ) {
            if (scan.hasNextDouble() ) {
                total = total + scan.nextDouble();
            }
            else {
                scan.next();
            }
        }
        scan.close();
        return total;
    } catch (IOException e) { UI.printf("File failure %s\n", e);}
}
```

Files with headers

- A data file may contain a header before the bulk of the data

  => need to read header first before reading rest of data

```
try {
    Scanner scan = new Scanner(Path.of(recordFileName ) );
    String name = scan.nextLine();
    int sID = scan.nextInt();
    String deg = scan.next();
    int count = scan.nextInt();
    for (int c = 0; c < count; c++){
        String code = scan.next();
        String grade = scan.next();
        int year = scan.nextInt();
        // process data
    }
    scan.close();
} catch (IOException e) { UI.println("File error: " + e); }
```

record-300765432.txt

```
Jo Miro
300765432
BSc
23
COMP102 A 2021
ENGR121 B+ 2021
COMP103 A- 2021
ENGR123 A- 2021
NZSL101  B+ 2021
COMP261 A- 2022
MATH261 B 2022
SWEN221 A+ 2022
  :
```

Files with multiple sets of data

```
try {
    Scanner scan = new Scanner(Path.of(recordFileName ) );
    while (scan.hasNext()){
        String name = scan.nextLine();
        int sID = scan.nextInt();
        String deg = scan.next();
        int count = scan.nextInt();
        for (int i=0; i < count; i++){
            String code = scan.next();
            String grade = scan.next();
            int year = scan.nextInt();
            // process data
        }
    }
    scan.close();
} catch (IOException e) { UI.println("File error: " + e); }
```

student-records.txt

```
Jo Miro
300765432
BSc
23
COMP102 A 2021
ENGR121 B+ 2021
COMP103 A- 2021
ENGR123 A- 2021
  :
SWEN221 A+ 2022
Jake Muskle
300765433
BA
16
```

Files with headers: passing a Scanner

- Can call another method to read the remaining data

  => must pass the Scanner to the method

```
Scanner scan = new Scanner(Path.of(recordFileName ) );
while (scan.hasNext()){
    String name = scan.nextLine();
    int sID = scan.nextInt();
    String deg = scan.next();
    int count = scan.nextInt();
    this.processRecord(scan, count, name, sID, deg);
}
scan.close();

public void processRecord(Scanner sc, int ct, String n, int ID, String deg){
    for (int i=0; i < ct; i++){
        String code = sc.next();
          // process data
    }
```

student-records.txt

```
Jo Miro
300765432
BSc
23
COMP102 A 2021
ENGR121 B+ 2021
COMP103 A- 2021
ENGR123 A- 2021
 :
SWEN221 A+ 2022
Jake Muskle
300765433
BA
16
```

Passing an open Scanner

- You can pass an open Scanner to a method
- The method can read some data from the Scanner
  - starts from wherever the Scanner had got up to
  - leaves the Scanner ready for other code to read from where it left off.

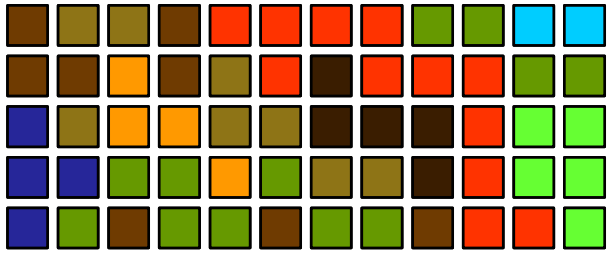
```
/** Reads and processes ct courses from the Scanner */
public void processRecord(Scanner sc, int ct, String n, int ID, String deg){
    for (int i=0; i < ct; i++){
        String code = sc.next();
          // process data
    }
}
```

student-records.txt

```
Jo Miro
300765432
BSc
23
COMP102 A 2021
ENGR121 B+ 2021
COMP103 A- 2021
ENGR123 A- 2021
 :
SWEN221 A+ 2022
Jake Muskle
300765433
BA
16
```

Files with headers

- Image files: ppm format



"Magic number" – code for ppm files

read into variables:
int cols
int rows

width (number of columns of pixels) and height (the number of rows of pixels)

maximum colour value

red-green-blue of each pixel, in turn

```
P3
12 5
255
200 182 163 215 198 177 130 116 93 37 28 9 31 22 7
81 67 38 83 71 42 6 5 6 0 0 0 57 68 60 97 112 104
97 92 76 202 186 165 97 82 60 32 25 5 38 30 13 103 90
63 158 140 97 58 49 25 43 42 17 107 104 74 127 140
113 95 102 79 66 58 41 71 57 37 41 30 7 82 71 41 111
95 64 174 157 120 115 101 63 49 43 12 67 65 30 126
124 74 133 136 97 88 87 62 98 93 54 78 63 37 108 93
62 121 104 69 135 120 88 190 172 139 36 30 15 1 0 0
16 17 9 64 77 58 50 57 39 7 2 0 105 106 64 121 103 71
117 100 67 159 144 113 212 197 171 161 146 114 0 0 0
0 0 0 37 48 32 72 88 68 24 26 19 12 12 9 74 72 49
```

nested for loops to
read colour of each pixel
set colour of UI
draw pixel.

More about static

```
/** Plot a graph of numbers from a file */
public class GraphPlotter {
    // Constants for plotting the graph
    public static final double GRAPH_LEFT = 50;
    public static final double GRAPH_RIGHT = 550;
    public static final double GRAPH_BASE = 400;
```

static means

"Belongs to class as a whole, Not to individual objects of this class."

final means

"Can't change the value once it has been set"

public means

"Can access this from code inside other classes"

private means

"Can only access this from code in **this** class"

Static methods: main

```
import ecs100.*;
import java.util.*;
import java.io.*;

public class GraphPlotter {

    :
    :

    public static void main(String[ ] args){
        GraphPlotter gp = new GraphPlotter();
        gp.setupGUI();
    }
}
```

main method
- static, because belongs to the class, not an object of the class
- called when the program is run directly from Java
- used when running a jar file

Using main

- Normally, you need a main method to be able to run your program.
- Typically, it creates an object of the class, and calls a method on the object.
- It can do more than that.

Why haven't we used main?

- BlueJ lets you create an object and call methods on it, using the mouse.
  - simpler methods
  - clearer understanding of objects and methods.
  - good for testing programs
  - ⇒ This course won't use main much,
    and will always be minimal.

Other static methods:

- Static methods are methods that don't need an object:
  - Methods in the Math class are static methods:

    Math.min(…)

    Math.max(…)

    Math.random()

    Math.sqrt(…)

  - Methods in the UI class are static methods:

    UI.drawRect(…)

    UI.println(…)

    UI.askInt(…)

None of these methods need an object to be created first.

Methods are called on the class itself, not on an object of that class.

Numeric data types

We have seen three types of numeric values

- int:
  - integer, with no fractional part (size = 32 bits)
  - eg: 75 -14532
  - range: -2,147,483,648 to 2,147,483,647
    -2^{31} to $2^{31} - 1$ or
    Integer.MIN_VALUE to Integer.MAX_VALUE

- long:
  - integer, but allows a bigger range (size = 64 bits)
  - eg: 7111333555L -123456789123456789L (L to say it is a long, not an int)
  - range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
    -2^{64} to $2^{64} - 1$
    Long.MIN_VALUE to Long.MAX_VALUE

Numeric data types

We have seen three types of numeric values

- double:
  - number with a fractional part. (size = 64 bits)
  - eg: 3.4 -193.0 -0.0063 4.8769e23 (= 4.8769×2^{23})
  - range: -2^{1024} to 2^{1024} or roughly -1.8×10^{308} to 1.8×10^{308}
  - precision: (accuracy) 15 decimal digits (precisely, 52 bits)
  - Special values:
    - Double.MAX_VALUE: largest positive finite value 1.797693e+308
    - Double.MIN_VALUE: smallest positive finite value 4.900000e-324
    - Double.NEGATIVE_INFINITY: double value smaller than any other double.
    - Double.POSITIVE_INFINITY: double value larger than any other double.
    - Double.NaN: "not a number": the error value (eg 1.0/0.0).

More numeric data types

We have seen two "wrapper" types of numeric values

- Integer:
  - wrapping up an int as an object so that it can be put into a list (for example)
- Double:
  - wrapping up a double as an object so that it can be put into a list (for example)

There are wrapper types for all the other numeric types.

Java will (in most cases) convert automatically between primitive and wrapper types.

Other numeric types

Integer types:

- byte (8 bits) -128 to 127
- short (16 bits) -32,768 to 32,767
  - Seldom used – just use int normally

Floating point:

- float (32 bits) smaller than doubles, less precision
  - eg 1.0f -0.4f
  - Seldom used, but sometimes needed for colours, eg Color.getHSBColor(0.4f, 1.0f, 1.0f);

Types and Coercion

- Mismatching types:

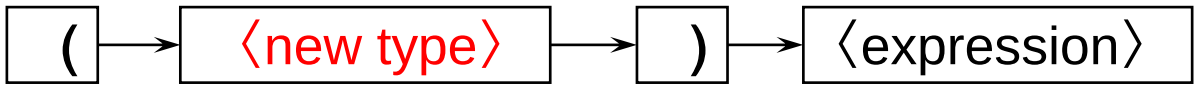
```
double num = scan.nextInt( );
int number = scan.nextDouble( );        ← Can't do this
double squareroot  =  Math.sqrt(25);    ← but sqrt wants double?
String name = "number-" + num;
```

- Java will "coerce" a value to the needed type if it can: eg
  - If a method needs a double and is given an int.
  - If a double variable is assigned an int value.
  - If "adding" any value to a String
  - converting between double and Double or int and Integer (or the other Wrapper Types)
- But only if it does not lose any information:
  - WON'T coerce a double to an int
  - WON'T coerce a String to a number, or vice versa (except when "adding" a number to a String)
  - WON'T coerce any object to a mismatching type (except when printing or "adding" to a String)

Casting

- Where it makes sense to convert a value into another type, but some information may be lost...
- You can *sometimes* "cast" the value to the other type:

```
int number = (int) Math.sqrt(49.5);
float red =  (float) Math.random();
```



- casting a double to an int will lose the fractional part and may mess up the value if the number is too big!
- Not everything can be cast to everything else!
  - ~~Scanner scan = (Scanner) (new PrintStream("data.txt"));~~