Keyboard and File IO

Class Scanner

- The Scanner class can read text from different sources:
  - Keyboard
  - File
  - String
  - …

Scanner (keyboard)

- Reading from the keyboard

```
Scanner scanner = new Scanner(System.in);

String line = scanner.nextLine();
```

  – Program blocks until user writes in the console and hits *enter*
  – The line introduced is kept in a String object.

Scanner (file)

```
Scanner scanner = new Scanner(new File("file.txt"));

String line = scanner.nextLine();
```

– It is mandatory to foresee the possibility of a non-existing file and decide how to deal with the situation.

Scanner (String)

- Token processing

```
String sentence = “one two    three four five“;
Scanner scanner = new Scanner(sentence);
int n = 0;
String inverted = “”;

while(scanner.hasNext()) {
   n++;
   String token = scanner.next();
   inverted = token + “ ” + inverted;
}
System.out.println(n + “ words”);
System.out.println(“Invertida: “ + inverted);
```

```
> 5 words
> Inverse: five four three two one
```

Files

Text

- Readable
- Extensive
- Format (nearly) universal
- Standard formats:
  - XML
  - SGML

Binary

- Not readable
- Compact
- Format may depend on machine architecture
- Standard formats:
  - ASN.1
  - Object Serialization Stream Protocol (Java)

Classes for text files

- `Scanner`
  - To read
  - Used before to read from classes
  - Establishes internal input *flow*

Object that is connected to the file and allows reading as a char sequence.

- `PrintWriter`
  - To write
  - Similar to `System.out`
  - Establishes internal output *flow*

- `File`
  - Represents a file

IO Exceptions

- What happens when
  - there is no such file?
  - data type does not match expected?
  - …

- An exeption is thrown

IO exceptions with Scanner

- `IOException`
  - `FileNotFoundException` – Attempt to establish a flow to to an IO device that does not exist.

Part of the program logic

- `RuntimeException`
  - `InputMismatchException` – Attempt to read does not match expected data type
  - `NoSuchElementException` – Attempt to read when File is finished
  - `IllegalStateException` – Attempt to read a closed File

Programming error! Reading must be recorded.

Exceptions for `PrintWriter`

- `IOException`
  - `FileNotFoundException` – Attempt to establish connection with a non existing File. Usually path problem, otherwise it would simply create a new one in most cases.

Part of Program logics

File access

- Open – Establish flow to/from file
- Interaction – Reading or Writing (sequentially)
- Close – Closing data-flow (optional in some cases) or exit

Read example: open

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import static java.lang.System.out;

…
try {
    final Scanner fileScanner =
        new Scanner(new File("My file.txt"));
    …
} catch (final FileNotFoundException exception) {
    out.println("File was not found. Sorry!");
    …
}
…
```

Scanner creates a data-flow to enter characters

Must deal with possible non-existing file!

Read example: read and close

Failing to read an int is a programming error , so a `RuntimeException` is thrown.

```
…
try {
    if (fileScanner.hasNextInt()) {
        final int numberOfCars = fileScanner.nextInt();
        …
    } else {
        out.println("Ops!");
    }
} finally {
    fileScanner.close();
}
…
```

Format errors must be dealt with explicitly!

An uncaught exception is propagated up the stack. The `Scanner` should be closed using a `finally` block.

Write example: open

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import static java.lang.System.out;

…
try {
    final PrintWriter fileWriter =
        new PrintWriter(new File("My new file.txt"));
    …
} catch (final FileNotFoundException exception) {
    out.println("Error creating file. Sorry!");
    …
}
…
```

`PrintWriter` creates a stream of character output.

Must deal with possible failure to open file (e.g., non existing folder, write access denied, etc.)

Write example: write and close

```
…
try {
    fileWriter.println(20);
    …
    if (fileWriter.checkError())
        out.println("Error writing to file.");
} finally {
    fileWriter.close();
}
…
```

Must deal with possible write errors.

Uncaught exception is propagated up through the stack. `finally` block used to close the file in any circumstance.

try-with-resources (Java 8.0)

try with auto-closable resources (implement AutoCloseable)

```
…

try (Scanner fileScanner = new Scanner(new File("My
   file.txt"));
     PrintWriter fileWriter =
         new PrintWriter(new File("My new file.txt"));
) {
   …
}  catch (FileNotFoundException e) {
   …
}
…
```

No finally necessary, resources auto-close when try finishes. Attention, read/write must happen inside try-block.

Object Streams

- Maintains relations between objects
- Write in binary format
- Easy to use
- ObjectOutputStream
- ObjectInputStream

Canais de Objectos

```
public class Aula implements Serializable {

   String nome = null;
   int n_presenças = 0;

   Disciplina disciplina = null;

   public Aula(String nome, int n, Disciplina d)
   {
       this.nome = nome;
       n_presenças = n;
       disciplina = d;
   }
// …
```

The Serializable interface

- To ensure persistency across program calls objects must be serializable
- All non-static (and non-transient) variables and objects are serializable
- The Serializable interface does not require any method implementation. Implementing object may be saved using writeObject() and read using readObject()

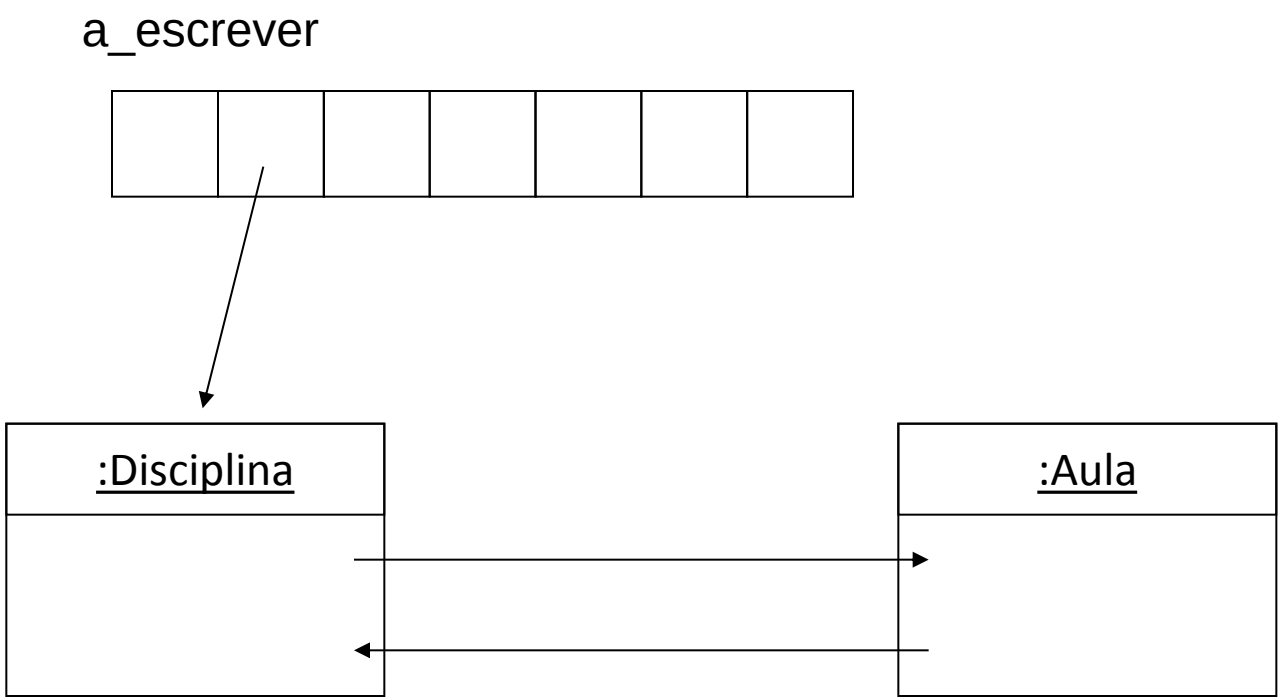
Object streams

```
public class Disciplina implements Serializable {


private String nome = null;
private int média_presenças = 0;
private Aula aula = null;

public Disciplina(String nome, String nome_aula, int
   n, int m) {
    this.nome = nome;
    média_presenças = m;
    aula = new Aula(nome_aula, n, this);
   }


// …
```

Object streams

```
public static void main(String[] args) {
  Disciplina[] a_escrever = new Disciplina[10];
  Disciplina[] a_ler = null;

  for (int i = 0; i != a_ler.length; ++i) {
      a_escrever[i] = new Disciplina(“D“ + i, “A“ +
  i,                    i*10, i*10 + 1);
  }
  // …
```

Object streams



Object streams

```
public static void main(String[] args) {
    // …
try {
   ObjectOutputStream o = new   ObjectOutputStream(new
                FileOutputStream("dat.dat"));
   o.writeObject(a_escrever);
   o.close();
} catch { //… }
try {
   ObjectInputStream in = new ObjectInputStream(new
   FileInputStream("dat.dat"));
   a_ler = (Disciplina[]) in.readObject();
   in.close();
} catch { //… }
```

Makes a deep copy with only one read/ write instruction.

Pre-defined streams

- Considered binary channels (although they transfer characters). Sub-classes of System.
- `System.in`: reads information from standard input (usually keyboard)
- `System.out`: writes to standard output, usually console
- `System.err`: special error stream, has priority over ***System.out*** (always flushes after writing)

System.in

```
import java.io.*;
// ...
BufferedReader teclado = new
    BufferedReader(new
  InputStreamReader(System.in));

String s = teclado.readLine();

System.out.println("Frase lida:\"" + s + "\"");
// ...
```

System.out

```
import java.io.*;
// ...
BufferedWriter monitor = new
    BufferedWriter(new
  OutputStreamWriter(System.out));
monitor.write("olá mundo");
monitor.flush();
```

Ou utilizar antes o método `print()` da classe `System.out`

```
System.out.println("olá mundo");
```

Sequencial Access: File

- Writing on the console the content of `files\info`:

```
import java.io.*;
// ...
int c;
File dados = new File("files","info");
FileInputStream f = new FileInputStream(dados);
While(c = f.read() != -1)
     System.out.print((char)c + ":");
```

Append mode

```
File dados = new File("info");

try {

   FileWriter f = new FileWriter(dados, true);
   f.write('A');
   f.close();

} catch (IOException e) {

   // ...

}
```

Dinamic Access

```
RandomAcessFile(File ficheiro, String modo)
```

- Read “r”;
- Write “w”;
- Read/write “rw”

Random Access, simultaneous read and write

```
File dados = new File(“..”,”info”);
RandomAcessFile f = new RandomAcessFile(dados, “rw”);

for(int i = 65; i < 91; i++)
   f.write(i) // escreve o alfabeto

byte[] ler = new byte[10];

f.seek(4); // salta para o quinto byte
f.read(ler, 0, 5); // lê as próximas 5 posições (E, F, G, H, I)

for(int = 0; i < 5; i++)
   System.out.println((char)ler[i] + “:”);
f.seek(3); // salta para o quarto byte (D)

System.out.println((char)f.read());
f.write(‘X’); // escreve ‘X’ por cima de ‘E’
```

Text-based formats

- CSV, TSV
- HTML / XML
- JSON
- …

JSON (simple)

object

{}
{ *members* **}**

members

pair
pair **,** *members*

pair

string **:** *value*

array

[]
[*elements* **]**

elements

value
value **,** *elements*

value

string
number
object
array
true
false
null

JSON (example)

```
{
    "linhas": [
        {
            "nome": "cozinha",
            "tomadas": [
                {"nome": "cozinha.1"},
                {"nome": "cozinha.2"},
                {"nome": "cozinha.3"}
            ]
        },
        {
            "nome": "quartos",
            "tomadas": [
                {"nome": "quartos.1"},
                {"nome": "quartos.2"},
                {"nome": "quartos.3"},
                {"nome": "quartos.4"}
            ]
```

```
 },
        {
            "nome": "sala",
            "tomadas": [
                {"nome": "sala.1"},
                {"nome": "sala.2"},
                {"nome": "sala.3"},
                {"nome": "sala.4"}
            ]
        }
    ]
}
```

JSON (example)

```
JSONArray linhasObj = (JSONArray) object.get("linhas");
for (Object obj: linhasObj) {
        nome = (String) ((JSONObject)obj).get("nome");
        JSONArray tomadasObj = (JSONArray)obj.get("tomadas");
        for(Object objt: tomadasObj) {
                JSONObject o = (JSONObject) objt;
                nome = (String) o.get("nome");
                tomadas.add(...);
        }
        linhas.add(...);
}
```

JSON (example)

```
JSONArray linhasObj = (JSONArray) object.get("linhas");
```

```
[
        {
            "nome": "cozinha",
            "tomadas": [
...
            "nome": "sala",
            "tomadas": [
                {"nome": "sala.1"},
                {"nome": "sala.2"},
                {"nome": "sala.3"},
                {"nome": "sala.4"}
            ]
        }
    ]
```

JSON (example)

```
JSONArray linhasObj = (JSONArray) object.get("linhas");
for (Object obj: linhasObj) {
        nome = (String) ((JSONObject)obj).get("nome");
        JSONArray tomadasObj = (JSONArray)obj.get("tomadas");
```

```
{
  "nome": "cozinha",
  "tomadas": [
     {"nome": "cozinha.1"},
     {"nome": "cozinha.2"},
     {"nome": "cozinha.3"}
   ]
}
```

```
[
  {"nome": "cozinha.1"},
  {"nome": "cozinha.2"},
  {"nome": "cozinha.3"}
]
```

References

- Java2 Platform API, Scanner, http://download.oracle.com/javase/6/docs/api/java/util/Scanner.html
- Java 2 Platform API, FileWriter, http://download.oracle.com/javase/1.4.2/docs/api/java/io/FileWriter.html
- Y. Daniel Liang, "Introduction to Java Programming" 7th Ed. Prentice-Hall, 2010.

Summary

- I/O (Scanner)
- Files