JAVA EXPERT COURSE

Lectures: 12 hours with 20 exercises

Requirements: Java Basic Course; SQL basic concepts.

Lecture	THEORY	PRACTISE
1	Object Oriented Programming: class types, attributes and methods (`static` and `final`).	None
2	Constructors, accessors and mutators. The `this()` constructor.	`Expert01`: overloading constructors. A simple call to `this()`.
3	Hierarchy: the `super()` constructor. Casting objects and testing the polymorphysm feature.	`Expert02`: `super()` and shadowing. `Expert03`: heritage, casting and back-casting.
4	Packages and access types. The `classpath` variable. Overriding access types.	`Expert04`: main class for testing. `Expert05`: sample “parent” class. `Expert06`: sample “child” class. `Expert07`: testing outside package. `Expert08`: extending outside package.
5	Managing streams: the `java.io` package. Writing, reading and formatting data on file.	`Expert09`: managing the `InputStream` and `OutputStream`.
6	JDBC Connections: executing SQL statements, defining JDBC Drivers, retrieving database meta information.	`Expert10`: basic JDBC connections. `Expert11`: executing a simple `Statement` and reading a `ResultSet`.
7	JDBC Statements: executing queries and reading data from a `ResultSet` object.	`Expert12`: creating tables and using the `executeBatch()` method.
8	JDBC Data: prepared statements, scrollable results, positioned updates, and data streams.	`Expert13`: prepared statements. `Expert14`: positioned updates. `Expert15`: reading data streams.
9	The AWT package: the *appletcation* and the `Frame` object: `paint()`, `repaint()`, `Canvas` and `FontMetrics` classes. Drawing pictures.	`Expert16`: a simple appletcation. `Expert17`: using `graphic.drawXXX()`.
10	Adding an AWT `Menu` object on a GUI. Managing the `Frame` object.	`Expert18`: a frame with a classical menu: File, Edit and Help.
11	Using the `FileDialog` object and the `File` class.	`Expert19`: opening a `FileDialog`. `Expert20`: using the `File` object.
12	Miscellaneus: OOD, the Garbage Collector and the JavaDoc tool.	None

1. *Object oriented programming (1h)*

- Types of classes (page 33, 34); attributes (page 35,36); using methods (page 37,38).
- Quick review of the methods implementation (page 39,40: already seen in Basic Java Course)

2. *Constructors (1h)*

- Review of the methods naming (page 41,42: topics left uncompleted in Basic Java Course).
- Constructors (page 47,48): the **`this()`** constructor.
- ➢ *Exercise*: write the **`Expert01`** application, which must have:
  - One array **`int[] dimensions`** as a class level attribute (having length = 2).
  - One constructor having signature **`Expert01(int[] dimensions)`**.
  - The **`main()`** method must create two objects: the first using **`new Expert01()`**, the second using **`new Expert01 (int[] dimensions)`**. If there are arguments for the **`main()`** method, the constructor must use these values to set the “dimensions” attribute.
  - The application must print the state (i.e. the “dimensions” value) of these two objects.

3. *Hierarchy*

- Hierarchy concepts (page 49,50): the **`super()`** method and the **`super`** qualification.
- Casting (page 51,52).
- ➢ Analyze the **`Expert02`** application, which extends `Expert01` and discuss shadowing.
- ➢ *Exercise*: write the **`Expert03`** application, which must define and create 3 helper classes: `Adam`, `Man` and `Woman`, where `Adam` is the parent class for `Man` and `Woman`. The children must override one parent’s method and introduce a new method. Then try all different possible casting.

4. *Packages and access types (1h)*

- Packages (page 53, 54): connections between **`package`** and the **`import`** statements.
- Access types (page 55, 56): discussing **`public`**, **`package`**, **`protected`** and **`private`** access.
- ➢ Analyze the **`Expert04`** application, which shows how different access types are available with respect to “child” or “not child” classes, and “same packages” or “not same packages” classes. Note that this application includes also the **`Expert05`** and **`Expert06`** classes.
- ➢ *Exercise*: write the **`Expert07`** application, which works as `Expert04` but for classes in a different package (*Hint*: create the **`Expert08`** class, extending `Expert06` in a different package).

5. *Managing streams*

- Managing data streams (page from 61 to 64): byte-based streams and character-based streams.
- Data Formatting (page 65, 66): the **`PrintWriter`** class and the portable data classes.

- ➢ *Exercise*: write the **`Expert09`** application. The application must read data from 1 input text file using 2 different systems: **`BufferedInputStream`** and **`BufferedReader`**. Then the application writes these data on 2 output files using **`BufferedOutputStream`** and **`BufferedWriter`**. Moreover, the application counts the number of steps of these input/output process, multiplies these 2 values, stores the result in a file using **`DataOutputStream`** and later retrieves it.

6. JDBC Connections

By calling **DriverManager.getConnection("**jdbc:<subprotocol>:<subname>**")**, the DriverManager class finds a Driver object, call the method Driver.connect() and return the connection (in this example "jdbc" is the protocol definition).
The subprotocol can specify the name of the mechanism (which could correspond to many drivers for the same DBMS) or the naming service (DNS, NIS or DCE) used to map to the actual DBMS.
The subname specifies the data source, using a syntax depending on the subprotocol, for example:
(Remark: only the ODBC subprotocol allows to specify parameters after the subname)

```
jdbc:<subprotocol>://<hostname>:<port>/<data source>
```

The Connection object allows 3 different ways to execute statements:

- **Statement**: created by Connection.createStatement(): a standard SQL query.
- **PreparedStatement**: created by Connection.prepareStatement(): useful to use many times the same statement, each time specifying different input parameters.
- **CallableStatement**: created by Connection.prepareCall(): useful to call DBMS stored procedure (server-side functions), handling many output parameters.

By default JDBC works in auto-commit mode, i.e. commit() is executed automatically after each statement. When using transactions (setting auto-commit to false) it is necessary to mind about transaction isolation levels, which define how to manage concurrency transactions. For example:

connection.**setTransactionIsolation**(TRANSACTION_READ_UNCOMMITTED);

Moreover the JDBC support the SQL3 paradigm, which implements User Defined data Types (UDT), Structured data types and DISTINCT data types. For more information see related documentation ("Using Type Maps").

➢ Analyze the **Expert10** application to have a first example of the JDBC paradigm.

Programmers can write new Driver classes, which must contain a static initializer that calls **DriverManager.registerDriver** passing an instance of the Driver as the parameter. In such a case, the best way to obtain this driver later is calling Class.forName("mypackage.Driver"). Alternatively, the Driver class could be added to the java.lang.System property jdbc.drivers:

jdbc.drivers=foo.Driver:steve.sql.Driver:morpheus.test.MyDriver (3 drivers)

In this case, the first call to a DriverManager method will automatically cause these driver classes to be loaded. Note that this second way of loading drivers requires a preset environment.
JDBC allows 3 ways to execute the statement object:

- **executeQuery()**: designed for statements producing a single result set, such as SELECT.
- **executeUpdate()**: for INSERT, UPDATE or DELETE, but also for Data Definition Language (DDL) statements such as CREATE TABLE, DROP TABLE or ALTER TABLE.
- **execute()**: statements that return more than one result set, more than one update count, or a combination of the two (advanced tecnique, seldom necessary).

➢ *Exercise*: write the **Expert11** application, which must query a simple table from an Access database, print column names and all rows in the table, retrieving any data as a String type.
Hint: define a static **display()** method to show data, it will be useful during next exercises!

7. 7. Reading data from Result Set

The `ResultSet` object is usually not updateble nor scrollable. This means that a `ResultSet` uses a cursor to point the current row, which can be moved only forward for many DBMS (i.e. `ResultSet.TYPE_FORWARD_ONLY` is the default). If the DBMS supports scrollable results, this cursor can be moved using: `previous`, `first`, `absolute`, `relative`, `afterLast` etc.

Remember that all methods for executing a statement close the current `Statement`'s result set (if open). This means that any processing of the current `ResultSet` object needs to be completed before a `Statement` object is re-executed. Here there are 3 different ways to count the number of rows (the 1st holds only for scrollable DBMS) :

```
rs.last();
int count=rs.getRow();
```

```
rs = stmt.executeQuery(
        "SELECT COUNT(*) FROM …");
rs.next();
int count = rs.getInt(1);
```

```
int count = 0 ;
while (rs.next()) {
    count++;
}
```

Rows are read using **`ResultSet.getXXX`** methods. To ensure portability with “forward only result sets”, values should be read from left to right and column values should be read only once.
The method **`getObject()`** will retrieve any data type, hence it is very useful when the information about the DB are uncomplete. Moreover this method is the only one that allows custom mapping. There are 3 possible cases about custom mapping:

- **`DISTINCT`** type with <u>standard</u> mapping: use the appropriate `getXXX()` method
- **`DISTINCT`** type with <u>custom</u> mapping: use the `getObject()` method
- **`SQL STRUCTURED`** type: use the `getObject()` method

Here it is the table of the recommented method for each data type:

TINYINT	`getByte()`	BIT	`getBoolean()`
SMALLINT	`getShort()`	CHAR	`getString()`
INTEGER	`getInt()`	VARCHAR	`getString()`
BIGINT	`getLong()`	LONGVARCHAR	`getAsciiStream()`
REAL	`getFloat()`	(ditto)	`getUnicodeStream()`
FLOAT	`getDouble()`	BINARY	`getBytes()`
DOUBLE	`getDouble()`	VARBINARY	`getBytes()`
DECIMAL	`getBigDecimal()`	LONG VARBINARY	`getBinaryStream()`
NUMERIC	`getBigDecimal()`	CLOB	`getClob()`
DATE	`getDate()`	BLOB	`getBlob()`
TIME	`getTime()`	REF	`getRef()`
TIMESTAMP	`getTimestamp()`	STRUCT	`getObject()`
ARRAY	`getArray()`	JAVA OBJECT	`getObject()`

`ResultSet` can have following attributes:

- `TYPE_FORWARD_ONLY`: nonscrollable; The view of the data depends on the DBMS.
- `TYPE_SCROLL_INSENSITIVE` : scrollable; it does not show changes to the underlying database that are made while it is open.
- `TYPE_SCROLL_SENSITIVE`: scrollable; if data are modified, the new values are visible, thus providing a dynamic view of the underlying data.

➢ *Exercise*: write the **`Expert12`** application, which must create 2 tables (PEOPLE and TASK) and populate these tables using the method **`statement.executeBatch()`**.

8. Updating data in the ResultSet

The `ResultSet` can have following concurrency types:

- `CONCUR_READ_ONLY`: this `ResultSet` *cannot* be updated programmatically and allows only read-only locks (i.e. there can be any number of concurrent users if the DBMS and driver allow it).
- `CONCUR_UPDATABLE` : this `ResultSet` *can* be updated programmatically and allows write-only locks, so that only one user at time has access to a data item.

The **`preparedStatement`** object it is useful when executing many times the same query with different parameters. To aknownledge if the DBMS supports these features use:

- `DatabaseMetaData.supportsResultSetType(int)`
- `DatabaseMetaData.supportsResultSetConcurrency(int, int)`
- `ResultSet.getType()`
- `ResultSet.getConcurrency()`

➢ See the **`Expert13`** application for some examples.

If the `ResultSet` is `CONCUR_UPDATABLE` it is possible to update, delete or insert data in the DB:

UPDATE

```
rs.updateXXX(<column>, <value>) ;
…

rs.updateRow() ;
```

INSERT

```
rs.moveToInserRow() ;
rs.updateXXX(<column>, <value>) ;
…
rs.updateRow() ;
rs.first() ;
```

Where the `column` index refers to the column number in the result set, not the column number in the database table, which is usually different (think about the `SELECT` syntax). Moreover, it is important to call **`updateRow()`** before moving the cursor to another row, otherwise any update will be lost. The update features are not supported by all DBMS, following requirements are often necessary:

- The query references only to a single table in the database
- The query does not contain a join operation or a `GROUP BY` clause
- The query selects the primary key of the table it references

Similar criteries holds for the **`insertRow()`** method:

- The user has read/write database privileges on the table
- The query selects all the nonnullable columns in the underlying table
- The query selects all columns that do not have a default value

Sometimes a DBMS present a `ResultSet` that seems to be `CONCUR_UPDATABLE`, but it is not!
In such a case the **`positioned updates`** methods may be used (see SQL specifications).

➢ *Exercise*: write the **`Expert14`** application, which must show all data from table PEOPLE and allow the user to choose and delete one row. Then this application must read data from the table TASK and look for `NULL` values.
➢ *Exercise*: write the **`Expert15`** application, which must test and compare the methods for retrieving very large row values, such as **`getBinaryStream()`** and **`getAsciiStream()`**.

1. Basic AWT

Using AWT it is possible to create an "appletcation", a mix of an applet and application.
To obtain this result just follow three steps:

1. Insert a `main()` method into the applet, to be run when the applet is used as an application.
2. Insert into the applet a subclass of `Frame`, which containes the applet as a class attribute.
3. If you want your frame to behave as a real window it must implement `WindowListener`.

We'll discuss about the **Frame** class in the next lecture ("Basic Menu's")

➢ Analyze the **`Expert16`** appletcation.

There are three main methods for managing the appearance of AWT's objects:

- **`paint()`**: here the programmer must implement all the code regarding the appearance.
- **`repaint()`**: called by the programmer when it is necessary to refresh the appearance.
- **`update()`**: called by the VM to refresh the appearance.

Notice that the programmer must never call **`paint()`** nor **`update()`** himself: anytime this is necessary the programmer must call **`repaint()`** instead. As soon as possible (considering other windows, other `repaint()` calls etc.) the VM will evaluate all pending `repaint()` calls and will call `update()` to actually refresh the component appareance.
In 99% of situations the programmer can forget about `update()`, but in certain cases it could be necessary to override this method to improve graphics performance.

Some quick hints/notes about some main AWT components:

- **`Canvas`**: to use a `Canvas` object is mandatory to create a child that overrides the `getMinimunSize()` and `getPreferredSize()` methods.
- **`FontMetrics`**: useful when using `drawString()` on AWT components (see page 46).
- **`Applet`**: comments about calling `init()` and playing audio files.

Using the API of the **graphics** class is possibile to draw many different objects: lines, rectangles, circles, arcs and polygons. For example, to draw a line just use `graphics.drawLine()`.
Notice that to draw a ticker line it is necessary to run a for-cycle in order to draw a set of parallel lines (each 1 pixel wide).

Showing images on a **Applet** is even easier, we need only following code:

```
Image image = getImage(URL address, "name.jpg") ;
graphic.drawImage(image, x, y, this) ;
```

Exercise: write the **`Expert17`** applet, which must draw or show the following figures:

- ♦ an empty green rectangle having round borders.
- ♦ a filled orange oval having black borders.
- ♦ a polygon drawing a red five-ends star.
- ♦ an image loaded from a JPEG file.

Moreover, the applet must contain a "Show" button. When this button is pressed, the JPEG image must disappear from the applet surface, and vicecersa (an on/off toggle behaviour).

2. *Basic Menu's*

AWT allows to create simple **G**raphic **U**ser **I**nterface providing standard "pull down" menu's.
In order to manage this system we need to learn about the hierarchy of the Java menu related items:

MenuBar: the main bar, using appearing horizontaly in the top part of the application's window.

- ♦ **Menu**: a single high-level menu entry of the top `MenuBar`, like "File", "Edit" or "Help".
    - ♦ **MenuItem**: a single entry of a `Menu`, like "New", "Open" or "Close" for the "File" menu.

An application usually has only one `MenuBar`, containing few `Menu` objects, but each `Menu` object can have more `MenuItem`'s, so the application as a whole will often have a lot of `MenuItem`'s.
We could say that, usually, there is one `MenuItem` object for each function offered to the user.
For this reason the `MenuItem` class provides the `addActionListener()` method, in order to react when the user clicks on the menu entry.

- ➢ Analyze the <u>first part</u> of the **`Expert18`** application, which is a simple AWT menù showing the standard menu entries like "File", "Edit" and "Help" but <u>not</u> providing any action.

Note that the main window containing the `Expert18` application is a **Frame** object, which implements the `WindowListener` interface. For this reason the `Expert18` class <u>must</u> define all the window's related methods, which are usually triggered by the user's actions.
For example, if the user clicks on the Windows® "Reduce to icon" button (on the top right corner of the window) the method `windowIconified()` will be executed. By this mechanism the default behaviour of the window can be completely changed.
Some useful methods of the **Frame** class are:

- `setSize()`: set the size of the window's frame (in pixels).
- `setLocation()`: set the location of the window's frame (in pixels).
- `toFront()`: brings the window's frame on the front of all other current windows.

In order to center the window in the middle of the screen is necessary to compare the screen size with the frame size. These information can be retrieved using following code:

```
Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
Dimension frameSize = frame.getSize();
int x = (screenSize.width - frameSize.width) / 2 ;
int y = (screenSize.height - frameSize.height) / 2 ;
frame.setLocation(x,y) ;
```

Another useful menù item is the **CheckboxMenuItem** class, implementing a menù item that can be selected or deselected like a standard checkbox. This item is different from the `MenuItem` since it provides the method `addItemListener()`. Morever, when an item is selected a "ticked" marker is showed on the left of the item's label, in order to "remember" the selection.

Exercise
Write the second part of the **`Expert18`** application, which must react to the "Open" entry by adding a new `Menu` object to the `editMenu`. This new menù must contain **3** `CheckboxMenuItem`'s allowing exclusive selection. Moreover, when this happens, all the `editMenu` entries ("Copy", "Cut" and "Paste") must be enabled.
Finally, if the user chooses the "Close" entry, the new menu must disappear (together with his `CheckboxMenuItem` items) and all the `editMenu` entries must be disabled.

3. Managing Files

It is possible to open a File Dialog panel using the **`FileDialog`** class, which has three constructors:

```
FileDialog dialog = new FileDialog(this, "Window Title") ;
FileDialog dialog = new FileDialog(this, "Window Title", FileDialog.LOAD) ;
FileDialog dialog = new FileDialog(this, "Window Title", FileDialog.SAVE) ;
```

The first two statements are equivalent, since the LOAD option is the default option: these file dialogs will allow only to load (i.e. "read") an existing file. The last statement opens a file dialog which allows to save a file on the hard disk.
When the `FileDialog` class is made visible (by calling **`setVisible(true)`**) the application "stops" waiting for an input from the user. For this reason, the next statement must be `fileDialog.`**`setVisible(false)`**.

- Analyze the **`Expert19`** application.

To handle system files Java provvides the **`File`** class. This class allow to get information about *real* files, as well to rename, move or delete them. The `Field` class introduces two very important fields:

```
static char separatorChar ;
static char pathSeparatorChar ;
```

The first field represents the character separating folder and files, for example "\" on Windows. The second field represent the path separator, which is ";" on Windows.
Using these static fields makes easier (and possible) to write portable code.
The **File** class works in the same way on <u>files</u> and on <u>folders</u>, for example the following code shows the content of the current folder, then it creates a new folder named "test".

```
File current = new File(".") ;
String[] list = current.list() ;
File subFolder = new File("test") ;
subFolder.mkDir() ;
```

- *Exercise*: write the **`Expert20`** application, which receives the `folder` String parameter using the `Expert19` class, then must print the following information about the files and the subfolders contained in the input `folder` parameter:
    - name of the file (or directory)
    - attributes: **d** for directory, **f** for file, **r** for readonly and **w** for writeble.
    - the date the object was modified the last time.
    - the length (size) in bytes of the object.

4. Miscellaneus

- Object Oriented Design: see page 58-59.
- The Garbage Collector: see page 45.
- The JavaDoc tool: see page 31-32.