14

المحاضرة الثامنة

The Java I/O System

إن هذا البحث من الأبحاث الهامة جداً، إذ أن تطبيقاته تمتد إلى مجالات واسعة لا تقتصر كما يظن البعض على الدخل والخرج النظامي والملفات، وسنترك هذه التفاصيل لتقوم المحاضرة بتوضيحها..

تأتي صعوبة هذا البحث من كثرة التفاصيل والـ classes الموجودة فيه، ولكننا إذا فهمنا الفكرة الأساسية للبحث فسنكون قد قطعنا أكثر من نصف الطريق وما بقي لنا إلا أن نقرأ عن تفاصيل هذه الصفوف في دليل اللغة.

ما هي الفكرة؟

قامت شركة *Sun* ببناء نظام I/O قوي جداً لـ Java -ولكنه معقد قليلاً- وضمنت جميع صفوفه في المكتبة (java.io)، وقد تطور هذا النظام عبر إصدارات الـ JDK المتعاقبة والتي تلت الإصدار الأول (JDK 1.0)، فقد حملت كل من النسختين (JDK 1.1، JDK 1.4) مجموعة من الصفوف الجديدة للتعامل مع نظام I/O.

الصف File:

قبل الدخول في تفاصيل نظام الـ I/O لابد من التطرق إلى class هام وهو (File).

إن اسم هذا الصف قد يوحي أنه قادر على إنشاء ملفات والقراءة منها والكتابة فيها، لكن مهمته في الحقيقة هي إعطاء معلومات شاملة عن الملفات والمجلدات.

عند إنشاء object من هذا الصف فإن الباني يأخذ مساراً معيناً (String)، هذا المسار قد يكون مسار ملف أو مجلد ويتيح مجموعة من العمليات على هذه الملفات.

سنستعرض مجموعة من التوابع المفيدة التي يمكن الاستفادة منها:

- isFile(): يرد true إذا كان المسار يمثل ملفاً.
- isDirectory(): يرد true إذا كان المسار يمثل مجلداً.

- list(): إذا كان المسار يمثل مجلداً فإن هذا التابع يرد مصفوفة تحوي أسماء جميع الملفات التي توجد في هذا المجلد.
- mkdir(): إذا كان المسار يمثل مجلداً غير موجود على القرص فإن هذا التابع يقوم بإنشائه.
- mkdirs(): إذا كان المسار يحوي مجموعة من المجلدات غير الموجودة على القرص فإن هذا التابع يقوم بإنشائها.

كما يحوي هذا الصف مجموعة كبيرة من التوابع الهامة التي تعطينا معلومات كاملة عن أي ملف، لذا أنصح زملائي بالاطلاع عليها.

مثال:

```
import java.io.*;

public class FileClass
{
    public static void main(String[] args) {
        File f = new File("C:\\myFolder");
        String[] fileList;

        if (f.exists()) {
            fileList = f.list();
            StringBuffer ss = new StringBuffer();
            for (int i = 0; i < fileList.length; i++)
                ss.append(fileList[i]);
            System.out.println(ss);
        }
        else
            f.mkdir();
    }
}
```

في المثال السابق اختبرنا كون المجلد (C:\myFolder) موجوداً على القرص، فإذا كان موجوداً سنطبع محتوياته، وإلا فسننشئه.

نظام الـ Input/Output:

- إن نظام الدخل/خرج في Java ممثل بمجموعة كبيرة من الصفوف التي تشترك مع بعضها بهرميات وراثية تطورت عبر نسخ JDK المتعاقبة.
- هذه الصفوف تقسم مناصفة بين القراءة والكتابة، أي أن كل صف للقراءة -تقريباً- يقابله صف للكتابة.
- إن التعامل مع مختلف أنواع الـ Streams (قراءة وكتابة) يتم بنفس الطريقة تماماً، وما يختلف فقط هو اسم الصف الذي نتعامل معه -كما سنرى بعد قليل-.
- فكرة التعامل مع نظام الـ I/O قائمة على مفهوم الطبقات:

لتتضح الفكرة أكثر سنضرب مثالاً من مادة الشبكات، حيث نعلم جميعاً أن البروتوكولات تقسم إلى عدة طبقات لكل منها مهمته الخاصة التي يعتمد فيها على الطبقة الأدنى منه مباشرة، ويستخدم ميزاتها بدون إعادة تعريف طريقة عملها ضمنه..

نفس الفكرة تنطبق على نظام الـ I/O في Java حيث تقسم صفوف القراءة والكتابة إلى عدة طبقات لكل منها مهمته الخاصة والتي يعتمد فيها على الطبقة الأدنى ويستفيد من توابعها وهذا مـا يـدعى بـالتغليف (**Wrapping**).

ما هو هذا التقسيم؟

كما في الشبكات إذ تكون الطبقة الدنيا مختصة بنقل البتات، لدينا مجموعة من الصفوف التي تختص بقراءة (وكتابة) البايتات، ولا تملك هذه الصفوف أي إمكانية للتعامل مع المعلومات الموجودة على الـ Streams مهما كان نوعها إلا كبايتات.

تأتي مجموعة أخرى من الصفوف التي تستطيع تمييز الـ primitives، ولكن هذه الـصفوف لوحـدها عاجزة عن إجراء أي عملية قراءة أو كتابة!!

وهنا يبرز مفهوم الطبقات حيث تعتمد هذه الصفوف على الطبقة الأدنى من الصفوف والتي تقـوم بقـراءة وكتابة البايتات، وتضيف إليها ميزاتها..

في الحقيقة لا يوجد تقسيم طبقي واضح، ولكن كل صف يعتمد على مجموعة من الصفوف الأدنى منه ويقدم خدمات جديدة، وهذه هي فكرة الدخل/خرج بأسرها، ولم يتبق علينا إلا أن نفصل فـي بعـض تقـسيمات الصفوف ونضرب أمثلة عنها ومن ثم نترك لكم مهمة التعرف على باقي الصفوف بحسب حاجاتكم..

صفوف المكتبة القديمة:

عندما ظهرت Java لأول مرة (JDK 1.0) كان نظام الـ I/O يقتصر على مجموعتين من الصفوف إحداهما للقراءة والأخرى للكتابة.

1. **القراءة**: إن جميع الصفوف المختصة بالقراءة مشتقة من الصف (**InputStream**)، وهو صف من النوع abstract ويحوي مجموعة من التوابع، وبالتالي فإن جميع أبنائه تحوي هذه التوابع، ومنها:
   - read(): يقوم هذا التابع بقراءة البايت التالي من الـ Stream.
   - read(*byte*[] b): وهو نسخة معدلة (overload) عن التابع السابق ويقوم بقراءة عدد مـن البايتات يساوي طول المصفوفة b ويضع هذه البايتات في المصفوفة.
   - close(): هام لإغلاق القناة بعد الانتهاء من القراءة خصوصاً إذا كانت مع ملف مثلاً.

سنستعرض مجموعة من الصفوف التي ترث من هذا الصف ونتعرف على مهمة كل منها:

❖ FileInputStream: ينشأ قناة مع الملف الذي ندخل مساره في الـ (constructor) تقوم هذه القناة بتأمين عملية القراءة من هذا الملف، ولكن القراءة تكون على شكل (bytes) بغض النظر عن نوع المعلومات الموجودة في هذا الملف.

❖ ByteArrayInputStream: يقوم بنفس العملية السابقة ولكن مع مصفوفة من الـ *bytes*.

❖ StringBufferInputStream: يقوم بنفس العملية السابقة ولكن مع String.

❖ PipedInputStream: يقوم بنفس العملية السابقة ولكنه يقرأ من ما يعرف بـ (Pipe)، وهو قناة تنشأ بين عدة تطبيقات قد تكون تابعة لنفس البرنامج (threads) أو تكون تابعة لعدة برامج على نفس الحاسب، وقد تكون هذه البرامج أيضاً على عدة حواسب (في حالة وجود شبكة).

❖ FilterInputStream: وله ابن هام جداً وهو:

- BufferedInputStream: يشكل مرحلة وسيطة في القراءة –كما سنرى في المثال–، ومهمته تنظيم القراءة.
- DataInputStream: نستطيع اعتبار هذا الصف ذا مستوى أعلى من الصفوف السابقة، إذ أنه غير قادر بمفرده على القراءة من أي نوع من الـ Streams ولكن الـ constructor يأخذ متحولاً من النمط InputStream[1] أي واحداً من أبناء هذا الصف الذين استعرضنا بعضهم قبل قليل، فإذا أدخلنا له مثلاً متحولاً من النمط FileInputStream فإنه يقرأ من ملف وهكذا..

  ميزة هذا الصف أنه يقدم توابع جديدة قادرة على قراءة الـ primitives منها: readByte(), readInt(), readDouble()...

  وهذا يعني أنه إذا غلف FileInputStream فسيقرأ من ملف، وإذا غلف PipedInputStream فسيقرأ من pipe وهكذا..

❖ ObjectInputStream: سنتكلم عن هذا الصف بالتفصيل عندما نتحدث عن قراءة الـ Objects..

2. **الكتابة**: جميع صفوف الكتابة مشتقة من الصف (**OutputStream**)، وهو صف من النوع abstract ويحوي مجموعة من التوابع، وبالتالي فإن جميع أبنائه تحوي هذه التوابع، ومنها:
   - write(*int* i): يقوم هذا التابع بكتابة i على الـ Stream.
   - write(*byte*[] b): وهو نسخة معدلة (overload) عن التابع السابق ويقوم بكتابة محتوى المصفوفة b على الـ Stream.
   - close(): هام لإغلاق القناة بعد الانتهاء من الكتابة خصوصاً إذا كانت مع ملف مثلاً.

   سنستعرض مجموعة من الصفوف التي ترث من هذا الصف ونتعرف على مهمة كل منها:

[1] سنطلق على العملية السابقة: تغليف النمط (InputStream) بالنمط (DataInputStream)

❖ FileOutputStream: ينشأ قناة مع الملف الذي ندخل مساره في الـ (constructor) إن كان موجوداً، وإلا فسينشئه وينشأ قناة اتصال معه، تقوم هذه القناة بتأمين عملية الكتابة على هذا الملف، ولكن الكتابة تكون على شكل (bytes).

❖ ByteArrayOutputStream: يقوم بنفس العملية السابقة ولكن مع مصفوفة من *bytes*.

❖ PipedOutputStream: يقوم بنفس العملية السابقة ولكنه يكتب على الـ (Pipe) الذي ذكرنا قليلاً عنه عندما تحدثنا عن الصف PipedInputStream.

❖ FilterOutputStream: وله ابنان هامان جداً وهما:

- BufferedOutputStream: صف هام جداً إذ أنه يمنع عملية الكتابة المباشرة على الـ Stream، ولكنه ينظم المعلومات عنده ولا يرسلها إلى الـ Stream إلا عند استدعاء التابع ()flush، وإن أهميته تبرز عند تغليف صف من مستوى عالٍ لصف من مستوى أدنى، وإن عدم استخدام هذا الصف كمرحلة وسيطة لا يضمن لك عدم تضارب المعلومات أثناء الكتابة.
- DataOutputStream: كما في مقابله من صفوف القراءة فإن هذا الصف ذو مستوى أعلى من الصفوف السابقة، إذ أنه غير قادر بمفرده على الكتابة على أي نوع من الـ Streams ولكنه يستطيع تغليف أي صف من النمط OutputStream ويقدم توابع جديدة قادرة على كتابة الـ primitives منها:

writeByte(), writeInt(), writeDouble()...

وهذا يعني أنه إذا غلف FileOutputStream فسيكتب على ملف، وإذا غلف PipedOutputStream فسيكتب على pipe وهكذا..

- PrintStream: ويقوم بنفس مهمات سابقه ولكنه يقدم التابعين print() و println().

❖ ObjectOutputStream: سنتكلم عن هذا الصف بالتفصيل عندما نتحدث عن كتابة الـ Objects..

ملاحظة:

إن معظم توابع الصفوف السابقة ترمي (throws) اعتراضات من النوع IOException، لذا فإن الـ compiler يجبرنا على معالجة هذا النوع من الاعتراضات.

مثال:

```
import java.io.*;

public class FileStreams
{
    public static void main(String[] args) {
        File myFile = new File("d:\\ammar.txt");
```

```
        // create the file and write in
        DataOutputStream d = null;
        try {
            d = new DataOutputStream(
                new BufferedOutputStream(
                new FileOutputStream(myFile)));
        }
        catch (FileNotFoundException ex) {
            System.err.println(ex);
        }

        try {
            for (int i = 'a'; i <= 'z'; i++)
                d.writeByte(i);
            d.flush();
        }
        catch (IOException ex1) {
            System.err.println(ex1);
        }
        finally {
            try {
                d.close();
            }
            catch (IOException ex2) {
                System.err.println("The file still opened!!");
            }
        }

        // read from the file
        DataInputStream in = null;
        try {
            in = new DataInputStream(
                new BufferedInputStream(
                new FileInputStream(myFile)));
        }
        catch (FileNotFoundException ex3) {
            System.err.println(ex3);
        }
        try {
            while (in.available() != 0)
                System.out.print((char)in.readByte() + " ");
        }
        catch (IOException ex4) {
            System.err.println(ex4);
        }
    }
}
```


لنشرح المثال السابق:

- وضعنا مسار الملف الذي سنتعامل معه ضمن غرض من الصف File.
- نريد أن نكتب primitives في هذا الملف لذا سنستخدم الصف DataOutputStream، والذي يجب أن يغلف صفاً من النوع FileOutputStream حتى نفتح الملف للكتابة إن كان موجوداً وننشئه إن لم يكن موجوداً، وسنستخدم تغليفاً وسيطاً بين الصفين السابقين وهو الصف BufferedOutputStream والذي ينظم عملية الكتابة كما سبق وذكرنا.

- إن بـــاني الـــصف FileOutputStream يرمـــي (throws) اعتراضـــات مـــن النـــوع FileNotFoundException.
- كل من التابعين ()writeByte و ()flush يرمي (throws) اعتراضات من النوع IOException.
- يجب أن لاننسى إغلاق الملف عن طريق التابع ()close والذي يرمي (throws) اعتراضات من النـوع IOException.[1]
- عند القراءة من الملف استخدمنا الصفوف المقابلة لصفوف الكتابة.
- التابع ()available يرد *true* عندما ينتهي الملف.

Readers & Writers:

أضافت Java في إصدارها التالي (JDK 1.1) مجموعتين جديدتين من الصفوف إحداهما للقـراءة والأخـرى للكتابة ولكنهما بقيتا في نفس المكتبة (java.io).

ولكن لماذا هذا التغيير؟

مشكلة الصفوف القديمة أنها تتعامل مع الـ *byte* (8-bit)، ولكن النظام العالمي تغيـر مـن ASCII إلـى Unicode والذي يعتمد ترميزاً من (16-bit) لذا كان لابد من الصفوف الجديدة التي تعتمد هذا النظام أي أنها تتعامل مع *char* بدلاً من *byte*، كما أن الصفوف الجديدة مصممة لتقوم بعملها أسرع من القديمة.

لا يمكن اعتبار المجموعة الجديدة بديلاً عن القديمة لأننا لا نستطيع الاستغناء عن القديمة، وإنما شكلت بعـض الصفوف الجديدة طبقة تعلو طبقة الصفوف القديمة وتغلفها.

1. **Reader**: وهو صف من النوع abstract ويشكل أباً لجميع الصفوف الجديدة المختصة بـالقراءة، ويحوي مجموعة من التوابع، وبالتالي فإن جميع أبنائه تحوي هذه التوابع، ومنها:
   - ()read: يقوم هذا التابع بقراءة المحرف التالي من الـ Stream (16-bit).
   - read(*char*[] cbuf, *int* offset, *int* length): يقرأ عدداً من المحـارف ويـضعها فـي المصفوفة.
   - ()close: هام لإغلاق القناة بعد الانتهاء من القراءة خصوصاً إذا كانت مع ملف مثلاً.

   سنستعرض مجموعة من الصفوف التي ترث من هذا الصف ونتعرف على مهمة كل منها:
   - ❖ StringReader: ويعتبر مقابلاً للصف StringBufferInputStream.
   - ❖ PipedReader: ويعتبر مقابلاً للصف PipedInputStream.

[1] إن استدعاء تابع ()close للـ DataOutputStream يستدعي تلقائياً توابع ()close لكل الصفوف المغلفة

- ❖ InputStreamReader: وهو صف هام ولكن ليس بحد ذاته وإنما لكونه الوسيط الوحيد بين صفوف المكتبتين القديمة والجديدة إذ أنه يستطيع تغليف صفوف المكتبة القديمة وتستطيع صفوف المكتبة الجديدة تغليفه، وبالتالي يمكن اعتباره طبقة وسيطة بين طبقتين، وله ابن وهو:
  - FileReader: ويعتبر مقابلاً للصف FileInputStream.
- ❖ BufferedReader: وهو أحد أهم صفوف هذه المكتبة إذ أنه يؤمن تعاملاً سهلاً ومريحاً مع الملفات النصية من خلال التابع readLine() الذي يقرأ سطراً كاملاً ويعيده كـ String، كما أن هذا الصف لا يقوم بالعمل بنفسه، وإنما يستطيع أن يغلف أي نوع من أنواع الصفوف Reader وبالتالي يستطيع القراءة من ملف أو من pipe أو...الخ

2. **Writer**: وهو صف من النوع abstract ويشكل أباً لجميع الصفوف الجديدة المختصة بالكتابة، ويحوي مجموعة من التوابع من بينها عدة نسخ للتابع write() يختص كل منها بكتابة نمط معين، ينصح بالاطلاع عليها، كما يحوي التابع close().

سنستعرض مجموعة من الصفوف التي ترث من هذا الصف ونتعرف على مهمة كل منها:

- ❖ PipedWriter: ويعتبر مقابلاً للصف PipedOutputStream.
- ❖ OututStreamWriter: وهو صف هام لكونه وسيطاً بين صفوف المكتبتين القديمة والجديدة إذ أنه يستطيع تغليف صفوف المكتبة القديمة وتستطيع صفوف المكتبة الجديدة تغليفه، وبالتالي يمكن اعتباره طبقة وسيطة بين طبقتين، وله ابن وهو:
  - FileWriter: ويعتبر مقابلاً للصف FileOutputStream.
- ❖ BufferedWriter: ومهمته هنا شبيهة بمهمة الصف BufferedOutputStream من المكتبة القديمة.
- ❖ PrintWriter: وهو صف هام جداً ومرن للغاية في التعامل مع الكتابة، حيث يقدم مجموعة كبيرة من النسخ لكل من التابعين print() و println() والتي تسهل عملية الكتابة، كما أنه يستطيع تغليف أي صف من عناصر كل من المكتبتين القديمة والجديدة.

مثال:

```
import java.io.*;

public class ReaderWriterIO
{
    public static void main(String[] args) {
        File myFile = new File("d:\\ammar.txt");

        // create the file and write in
        PrintWriter p = null;
        try {
            p = new PrintWriter(new BufferedWriter(new FileWriter(myFile)));
        }
```

```
        catch (IOException ex) {
            System.err.println(ex);
        }
        p.println("Hello world");
        p.println("I love Java!");
        p.close();

        // read from file
        BufferedReader b = null;
        try {
            b = new BufferedReader(new FileReader(myFile));
        }
        catch (FileNotFoundException ex1) {
            System.err.println(ex1);
        }
        String s;
        try {
            StringBuffer buf = new StringBuffer();
            while ((s = b.readLine()) != null)
                buf.append(s + "\n");
            System.out.println(buf);
        }
        catch (IOException ex2) {
            System.err.println(ex2);
        }
        finally {
            try {
                b.close();
            }
            catch (IOException ex3) {
                System.err.println(ex3);
            }
        }
    }
}
```


- نلاحظ أننا استخدمنا الصف BufferedWriter كمرحلة وسيطة بين الصفين FileWriter وPrintWriter وذلك لأن هذا الـ (wrap) يزيد من الفعالية وينظم الكتابة.
- في مرحلة القراءة عندما ينتهي الملف فإن تابع readLine() يرد *null* وعندها نستطيع معرفة أن الملف انتهى.
- نلاحظ أننا أضفنا المحرف "\n" إلى نهاية كل سطر نقرأه من الملف وذلك لأن التابع readLine() لا يرد محرف آخر السطر.

RandomAccessFile:

تكتسب الملفات أهمية خاصة من بين عناصر الدخل والخرج، إذ أننا نستطيع التعامل معها بطرق أخرى مختلفة عن الطريقة التتابعية التي كنا نستخدمها حتى الآن، حيث تتعامل جميع الصفوف التي استعرضناها حتى الآن مع الدخل والخرج بطريقة مرتبة ولا تمكننا من الكتابة أو القراءة في أماكن عشوائية أو القفز ضمن الملف مثلاً..

من الواضح أن الطريقة السابقة لا تلبي جميع احتياجات المبرمج إذ أنه قد يحتاج إلى النفاذ بشكل عشوائي للملف[1]، ومن هنا ظهرت الحاجة للصف: RandomAccessFile.

الغريب في هذا الصف أنه لا يتبع لأي من الهرميات السابقة التي ذكرناها، وإنما يقف وحيداً ضمن المكتبة java.io، حتى أنه لا يشترك مع صفوف المكتبتين السابقتين بأي شيء بل يقدم توابعه بشكل مستقل تماماً.

يمكن فتح الملف للقراءة فقط أو للقراءة والكتابة عن طريق هذا الصف، وذلك بتمرير إحدى القيمتين التاليتين للباني: "r" أو "rw".

يمتلك هذا الصف مجموعة من التوابع الهامة والتي سنتكلم عن بعضها:

- getFilePointer( ): يرد المكان الحالي للمؤشر ضمن الـ file[2] مقدراً بالـ byte.
- seek(long pos): ينقل المؤشر إلى الـ *byte* (pos) من الملف.
- length(): يرد طول الملف مقدراً بالـ *byte*.
- بالإضافة إلى مجموعة متميزة من توابع القراءة والكتابة التي يمكنكم الاطلاع عليها..

Standard I/O:

تعاملنا مسبقاً مع الخرج النظامي حيث استخدمنا العبارة System.out.println()، وآن الأوان لنفهم ماهية عمل هذه العبارة:

في الحقيقة فإن الصف System يحوي الحقل التالي:

```
public final static PrintStream out;
```

وقد تحدثنا سابقاً عن الصف PrintStream وذكرنا أنه يملك كلاً من التابعين print() و println().

نلاحظ أن Java قدمت لنا خرجاً نظامياً من مستوىً عالٍ، ولكن ماذا عن الدخل النظامي؟

قدمت Java دخلاً نظامياً وهو الغرض (System.in)، ولكن المشكلة أن هذا الغرض من النوع InputStream أي أنه من المستوى الأدنى والذي لا يستطيع التعامل إلا مع الـ *byte*.

لذا لن نستطيع التعامل مع الدخل النظامي بالمرونة المطلوبة إلا إذا غلفنا هذا الغرض بأغراض من صفوف تتمتع بقدرات أكبر..

أفضل صف يمكن أن يغلف الدخل النظامي هو BufferedReader، ولكن المشكلة تكمن في أنه لا يغلف إلا الصفوف التي ترث من Reader، لذا نحتاج إلا صف وسيط بين المستويين السابقين وهو InputStreamReader، الذي يستطيع أن يغلف الغرض System.in ومن ثم يغلفه

---

[1] تعرف هذه العملية بـ seek

[2] طبعاً من المعروف أننا نتجول ضمن الملف عن طريق مؤشر

BufferedReader، وعندها نستطيع القراءة من الدخل النظامي عن طريق التوابع التـــي يتيحهـــا الـــصف BufferedReader.

كما يمكننا تحويل الدخل والخرج النظاميين إلى نوع الـــ Stream الذي نريده عن طريق التوابع:

setIn(InputStream)
setOut(PrintStream)

مثال:

```
import java.io.*;

public class RandomAccess
{
    public static void main(String[] args) {
        File myFile = new File("D:\\myFolder");
        BufferedReader stdin = new BufferedReader(
                               new InputStreamReader(System.in));
        RandomAccessFile rand = null;
        try {
            rand = new RandomAccessFile(myFile, "rw");
        }
        catch (FileNotFoundException ex) {
            System.err.println(ex);
        }

        try {
            String s = stdin.readLine();
            rand.seek(4);
            rand.writeChars(s);
        }
        catch (IOException ex1) {
        }
        finally {
            try {
                stdin.close();
                rand.close();
            }
            catch (IOException ex2) {
                System.err.println(ex2);
            }
        }
    }
}
```


Object serialization:

- حتى نستطيع كتابة الـ Objects على مختلـف أنـواع الـــ Streams يجـب أن نـستخدم الـصف ObjectOutputStream.
- هذا الصف يغلف أحد الصفوف OutputStream وبالتالي يستطيع كتابة الـ objects علـى مختلـف أنواع الـ Streams عن طريق التابع ()writeObject.

- لنستطيع كتابة أي object عن طريق الصف السابق يجب أن يكون ذلك الصف محققاً لأحد الـ interfaces التالية: (Serializable) أو (Externalizable).
- نستطيع قراءة هذه الـ object من جديد عن طريق الصف ObjectInputStream الذي يغلف أحد الصفوف InputStream وبالتالي يستطيع قراءة الـ objects من مختلف أنواع الـ Streams عن طريق التابع readObject().
- إن مشكلة قراءة الـ objects بهذه الطريقة أننا لن نستطيع تمييز نوعه، إذ أن التابع readObject() يرد مؤشراً من النوع Object وعلينا أن نقوم بعملية Down-casting حتى نعيده إلى نمطه الأصلي، وهذا يقتضي معرفتنا التامة للتنسيق الذي كتب به الملف.
- ماذا نعني بكتابة الـ object؟

في الحقيقة إن كتابة الـ object على Stream ما تتضمن كتابة جميع حقوله. لا مشكلة حتى الآن .. ولكن ماذا إذا كان هذا الـ object يحوي objects في داخله (composition)؟

- الميزة الهامة هي أن الـ object لا يكتب وحده، وإنما تكتب جميع حقوله الـ primitives و الـ objects معه وحتى لو كان كل object منها يحوي في داخله عدداً من الـ objects فستكتب جميعاً.
- ملاحظة: الحقول الـ static لا تكتب مع الـ object وإنما تعطى قيماً صفرية.
- ماذا لو لم أشأ أن أكتب جميع الحقول؟

لنفرض أن لدينا حقلاً يحوي كلمة المرور (password) ولم أرد أن أكتبه على الـ pipe مثلاً خوفاً من أن يستطيع أحد البرامج التنصت على عملية النقل وكشف محتوى الطرود، فما الحل؟

أتاحت لنا Java هذه الميزة وذلك بوضع الكلمة المحجوزة *transient* قبل الحقل الذي لا أريد كتابته، وبالتالي لن يكتب هذا الحقل وعند الاسترجاع سنجد أن قيمته عادت إلى القيمة الصفرية.

مثال:

```
import java.io.*;

class Data implements Serializable
{
    private int sn;
    private transient String password;

    public Data(int sn, String password) {
        this.sn = sn;
        this.password = password;
    }

    public String toString() {
        return "My sn = " + sn + "\nMy Password: " + password;
    }
}
```

```
public class Student implements Serializable
{
    Data d = new Data(3, "123456");
    String name;

    public Student(String name) {
        this.name = name;
    }

    public String toString() {
        return "My Name: " + name + "\nMy Data: \n" + d;
    }

    public static void main(String[] args) {
        Student s1 = new Student("Ammar");
        ObjectOutputStream output = null;
        try {
            output = new ObjectOutputStream(new FileOutputStream("c:\\myFile"));
            output.writeObject(s1);
        }
        catch (IOException ex) {
            System.err.println(ex);
        }

        ObjectInputStream input = null;
        try {
            input = new ObjectInputStream(new FileInputStream("c:\\myFile"));
            Student s2 = (Student)input.readObject();
            System.out.println(s2);
        }
        catch (ClassNotFoundException ex1) {
            System.err.println(ex1);
        }
        catch (IOException ex2) {
            System.err.println(ex2);
        }
    }
}
```

إن خرج البرنامج السابق هو:

```
My Name: Ammar
My Data:
My sn = 3
My Password: null
```

نلاحظ أن المثال السابق يوضح لنا جميع المفاهيم التي تحدثنا عنها، وننتبه إلى أن قيمة الـ password كانت null لأنها لم تكتب أصلاً في الملف وذلك بسبب الكلمة (*transient*).

في نهاية هذه المحاضرة لا يسعني إلا أن أقدم اعتذاراً لكم على تأخيرها، فأنتم تعلمون مدى الضغط الدراسي الذي نعاني منه، وأريد أن أنوه إلى أنني التزمت في هذه المحاضرة بما أعطاه الدكتور تماماً، ولم أستطع تغطية جميع الأفكار التي يحويها البحث –كما عودتكم من خلال المحاضرات السابقة– وذلك لسببين:

1. الضيق الشديد في الوقت.
2. إن الدكتور لم يعط أكثر من ثلث البحث تقريباً، لذا فإن الإحاطة بباقي الأفكار مهمة شاقة وخصوصاً أنها غير مطلوبة في الفحص، لذا أنصح زملائي الأعزاء بقراءة البحث من المرجع لأن فيما تبقى من البحث الكثير من الأفكار الممتازة –خصوصاً إذا أخبرتكم عن وجود نسخة ثالثة من صفوف الـدخل والخـرج تدعى بـ New Input Output (nio) تتميز بالفعالية والوثوقية العالية[1]–.

انتهت المحاضرة

[1] أحببت أن أشوقكم لقراءة البحث