Laborator 7

Serializarea si fluxurile de date

I/O Streams

La baza oricaror operatii de intrare/iesire in java se afla un Stream. Exista douar doua tipuri de streamuri, InputStream si OutputStream.
Observam ca acestea doua sunt de fapt interfete. De ce?
Ce este un stream? Streamul, tradus pe romaneste “flux” este exact ceea ce se intelege, adica un flux continuu de date, intr-o directie (dinspre noi -> OutputStream, spre noi -> InputStream). Trebuie sa ne imaginam aceste streamuri ca niste tevi in care sunt introduse bucati de lego, respectiv din care sunt scoase bucati de lego.

Vorbim acum de streamurile de intrare, InputStream.
Nu avem neaparat o garantie a faptului ca un stream se va termina (va ajunge la sfarsit) si in mod cert nu putem sti de la inceput cand urmeaza sa se termine pana nu parcurgem (adica cititm) tot streamul. Mai mult odata extrasa informatia, nu putem sa o introducem la loc.

Vorbim acum despre streamurile de iesire, OutputStream.
Acum ne aflam la celalalt capat al tevii, si putem sa introducem date. Ceea ce nu putem sa facem este sa luam inapoi ceea ce am introdus, sau sa editam ce am introdus.

Ambele fluxuri au metode pentru a putea scrie/citi elemente absolut fundamentale in/din ele. De cele mai multe ori ne va fi dificil sa lucram cu ele.

Low Level/High level streams

Deja stim ca nu putem sa construim o clasa abstracta, clasa abstracta este doar o promisiune. Deci pentru a putea lucra cu un stream trebuie sa il luam de la cineva sau sa construim o clasa care implementeaza un stream. Pentru sarcinile uzuale exista diverse clase care extind Input sau Output stream. De exemplu, citirea/scrierea din/in fisier (observati denumirea)

FileInputStream
FileOutputStream

Aceste obiecte extind Input/OutputStream si au un constructor. Cel mai simplu mod de a obtine un stream pentru a citi/scrie intr-un fisier este:

```
InputStream fis = new FileInputStream(“fisieri.txt”);
OutputStream fos = new FileOutputStream(“fisiero.txt”);
```

Observati ca nu am pastrat tipul fluxului pentru ca de fapt nu ne intereseaza.
Exista si alte obiecte care extind InputStream si OutputStream, puteti sa vedeti aici:
https://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html
Ceea ce ne intereseaza insa pe noi este faptul ca aceste clase extind direct cele doua interfete. Din aceasta cauza le numim “low level” streams, adica ele se ocupa strict cu punerea de date (piese lego singulare) in stream (teava).

Evident ca poate fi foarte greu sa scriem byte cu byte datele noastre intr-un fisier. Pentru a ne ajuta sa facem acest lucru, au fost definite alte clase, de exemplu:

DataInputStream/DataOutputStream (pentru citirea/scrierea de primitive)
PrintWritter/InputStreamReader (pentru lucrul cu stringuri)
Scanner (un caz special pentru lucrul cu stringuri)
ObjectInputStream/ObjectOutputStream (pentru citirea/scrierea de obiecte).

Toate aceste obiecte sunt niste “masti” sau “adaptoare” care se monteaza la un capat sau celalalt al unui stream. Imaginati-va ca vreti sa puneti in teava nu doar simple piese de lego, ci vreti sa puneti diverse constructii. Voi dati acestui adaptor o casuta, el o descompune in piese (bytes) pe care le pune in teava (stream) iar apoi la celalalt capat cel care primeste piesele are optiunea sa foloseasca adaptorul pereche pentru a lua din teava (stream) piesele (bytes) si sa reasambleze casuta.

Deoarece aceste obiecte sunt doar niste masti, ele au nevoie ca sa lucreze de un low-level stream la care sa se ataseze. Deci vom avea constructori pentru aceste obiecte care primesc ca parametru un low-level stream (input sau output). Aici devine important rolul interfetelor Input/Output Stream. Putem practic sa folosim orice fel de stream (fisier, retea, audio etc) fara sa stim exact de unde isi trage el datele!
Aceste obiecte care se construiesc pe baza unor low-level streams sunt High-Level streams.

Procesul prin care un obiect este descompus in parti mai mici, atomice, se numeste serializare. Acest proces este (trebuie sa fie!) reversibil, iar operatia inversa se numeste deserializare.

Exercitiu 1:

Explorati clasele DataInputStream si DataOutputStream
Explorati PrintWritter si InputStreamReader
Vedeti ce metode aveti la dispozitie.

Object Input/Output Stream si serializarea in Java

Clasele ObjectInputStream si ObjectOutputStream sunt un caz special pentru ca ele ne scutesc de foarte multa bataie de cap cand este vorba de serializarea si deserializarea obiectelor.

Sa spunem ca avem o clasa Profesor, care extinde Persoana, si mai multe obiecte care au acest tip. Vrem sa le scriem intr-un fisier.
Cea mai simpla solutie ar fi urmatoarea:

1. Clasa Persoana implements Serializable (nu trebuie sa definim nimic suplimentar). Clasa Profesor este si ea Serializable?
2. Obtinem un stream de iesire (new FileOutputStream)
3. Construim un ObjectOutputStream pe baza streamului low level de iesire.
4. apelam pe rand, pentru fiecare persoana, metoda “writeObject” din ObjectOutputStream.
5. Nu uitam sa inchidem streamul!

Gata! Obiectele au fost serializate!

Ceea ce trebuie sa intelegem este insa ca aceste obiecte au fost serializate folosind mecanismul standard de serializare in Java. Adica niste baieti destepti au scris cod care este capabil sa ia orice obiect de-al nostru si sa il converteasca in un sir de bytes pe care apoi sa poata sa il converteasca la loc intr-un obiect similar.

Pentru ca acest mecanism sa functioneze, tot ce trebuie sa facem este sa ne asiguram ca obiectul pe care vrem sa il salvam este Serializable, si toate campurile care il definesc sunt si ele Serializabile la randul lor (iar conditia se aplica recursiv). Desigur, toate primitivele sunt Serializable (chiar daca nu implementeaza interfata, evident) si in plus si clasele uzuale din java (in general, daca are sens sa fie Serializable o clasa, cel mai probabil este). Puteti verifica fiecare clasa in particular folosind documentatia Java. Daca aceasta regula este incalcata, vom primi **la runtime** o exceptie.

Sa spunem ca in Persoana avem un camp static. Acest camp static NU va fi serializat. De ce? Pentru ca acel camp nu defineste o persoana anume, ci este ceva generic, deci nu are sens sa il serializam (la fel cum nu are sens sa il deserializam).
Sa spunem ca in Persoana vrem sa avem un camp care sa nu fie serializat. Putem marca acel camp ca fiind “transient”, iar acest lucru spune mechanismului standard de serializare al Java ca acel camp nu trebuie sa fie serializat.
Mare atentie: toate celelalte campuri vor fi serializate (inclusiv cele private). De altfel daca nu s-ar serializa campurile private, si noi deja suntem obisnuiti sa avem DOAR campuri private, nu ar fi inutila serializarea?
Un alt lucru important de mentionat este ca atunci cand deserializam un obiect constructorul NU este apelat!

Serializare “custom” in Java

In vasta majoritate a cazurilor serializarea standard pusa la dispozitie este suficienta dar sa spunem ca vrem sa facem si altceva, ceva special, dupa serializare/deserializare. De exemplu, vrem ca inainte de serializarea varstei persoanei sa o “criptam” folosind o operatie XOR. Evident acelasi lucru trebuie sa il facem cand vrem sa o deserializam.

Pentru a face acest lucru implementam in clasa Persoana metodele readObject si writeObject care primesc ca parametru un ObjectInputStream respectiv un ObjectOutputStream.
Aceste metode se vor executa IN LOCUL implementarii default. Dar pentru a face ceea ce ne-am propus noi trebuie sa avem la dispozitie si implementarea default inainte/dupa care sa facem noi ceva special.
Pentru a avea acces la implementarea default, folosim defaultWriteObject sau defaultReadObject din cele doua streamuri pe care le primim ca parametru.

Serializare “custom” metoda 2

Exista o interfata, numita Externalizable.
Aceasta interfata declara doua metode, readExternal si writeExternal care au ca parametrii un obiect de tipul ObjectInput respectiv ObjectOutput. Acestea sunt obiecte distincte fata de ObjectInputStream si ObjectOutputStream, dar puteti sa le folositi in mod similar, deci pentru simplitate haideti sa le consideram identice.
Daca un obiect implementeaza Externalizable, este datoria lui sa scrie in stream ce trebuie pentru a putea sa fie deserializat. La fel, este datoria lui sa citeasca ce trebuie (si doar ce trebuie!) din stream pentru a isi reface starea initiala.

Observam cateva diferente:

1. Apelul objectOutputStream.writeObjecT(myObject) comparat cu myObject.writeExternal(objectOutputStream) (si analog pentru read)
2. pentru a chema myObject.readExternal trebuie sa construim obiectul de tipul corespunzator si apoi sa ii citim “starea” de pe stream. Altfel, putem sa citim “un obiect” de pe stream care de fapt este ceea ce vrem noi sa fie.

Serializarea text sau binar

Serializarea este un concept care presupune transformarea unor date complexe in date primitive ce pot fi trimise sau salvate usor.
In functie de ceea ce intelegem prin date primitive, putem avea serializarea in format binar (ca bytes) sau text (ca siruri de caractere).
Atentie! Cel mai simplu exemplu, daca vrem sa serializam sirul “123” ca binar si ca text vom observa doua rezultate complet diferite!
Exemplul clasic de serializare ca text a unui obiect este serializarea ca XML sau ca JSON, ambele fiind folosite pe scara larga.

Exercitiu 2

Serializati/Deserializati o persoana (profesor) ca Text

Exceptiile

Ce sunt exceptiile?

Fundamental, exceptiile sunt de fapt un al doilea return type al unei functii, si sunt instante al unui tip Throwable sau al unei sublclase a lui. Ele sunt un mod de a comunica apelantului ca ceva nu s-a intamplat normal in executia metodei si metoda nu si-a putut incheia activitatea sau nu a putut sa returneze rezultatul promis.
Exista doua categorii mari de exceptii, “treated exceptions” si “untreated exceptions”
Exceptiile trebuie in general tratate intr-un fel sau altul, fie explicit intrun “catch” fie sunt aruncate mai sus.

Exceptiile netratate (untreated)

Aceste exceptii sunt instante de RuntimeException. RuntimeException extinde Exception care extinde la randul ei Throwable.
Ce le face sa fie speciale este faptul ca ele nu trebuie tratate. Exceptiile de acest tip pot practic sa apara oriunde iar tratarea lor explicita (cu un catch) ar incarca enorm codul, de cela mai multe ori inutil.
Ce exceptii netratate cunoasteti?

Exceptiile tratate (treated)

Aceste exceptii sunt….restul. Orice exceptie nu este un RuntimeException este automat o exceptie tratata.

Reguli pentru aruncarea si tratarea exceptiilor

Aceasta discutie este relevanta doar in interiorul unei functii.
Exista doua metode prin care putem sa ne lovim de o exceptie:

1. facem throw new Exception() sau ceva similar
2. o operatie din interiorul metodei a aruncat in interior o exceptie.

In acest caz avem doua optiuni: fie tratam exceptia (cu un catch) fie o aruncam mai sus (adica nu avem un catch) si in acest caz metoda noastra trebuie sa declare ca throws exceptia respectiva.