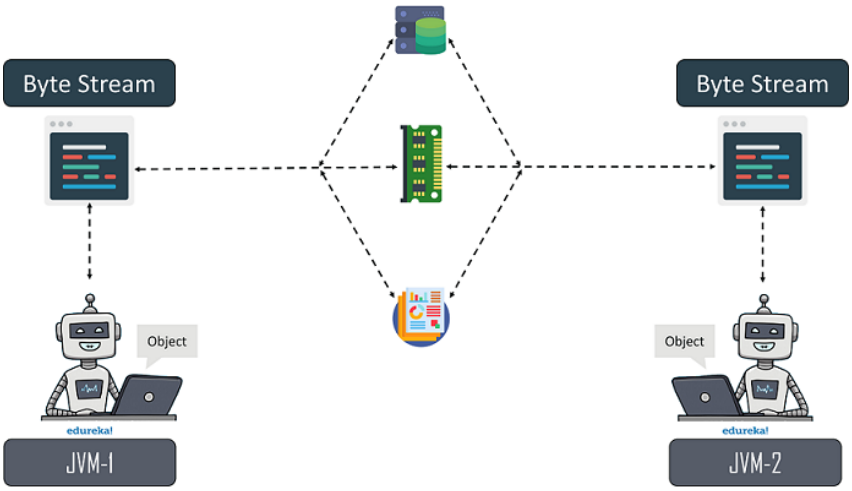
SERIALIZAREA OBIECTELOR

Ciclul de viață al unui obiect este determinat de executarea programului, respectiv obiectele instanțiate în cadrul său sunt stocate în memoria internă, astfel încât, după ce acesta își termină executarea, zona de memorie alocată programului este eliberată. În cadrul aplicațiilor, de cele mai multe ori, se dorește salvarea obiectelor între diferite rulări ale programului sau se dorește ca acestea să fie transmise printr-un un canal de comunicație. O soluție aparent simplă pentru rezolvarea acestei probleme ar constitui-o salvarea stării unui obiect (valorile datelor membre) într-un fișier (text sau binar) și restaurarea ulterioară a acestuia, pe baza valorilor salvate, folosind un constructor al clasei. Totuși, această soluție devine foarte complicată dacă unele date membre sunt referințe către alte obiecte, deoarece ar trebui salvate și restaurate și stările acelor obiecte externe! Mai mult, în acest caz nu s-ar salva funcționalitățile obiectului (metodele sale) și constructorii.

Limbajul Java permite o rezolvare simplă și eficientă a acestei probleme prin intermediul mecanismelor de *serializare* și *deserializare* (sursa imaginii: https://www.edureka.co/blog/serialization-in-java/):



Serializarea este mecanismul prin care un obiect este transformat într-o secvență de octeți din care acesta să poată fi refăcut ulterior, iar *deserializare*a reprezintă mecanismul invers serializării, respectiv dintr-o secvență de octeți serializați se restaurează obiectul original.

Utilizarea mecanismului de serializare prezintă mai multe avantaje:

- obiectele pot fi salvate/restaurate într-un mod unitar pe/de pe diverse medii de stocare (fișiere binare, baze de date etc.);
- obiectele pot fi transmise foarte simplu între mașini virtuale Java diferite, care pot rula pe calculatoare având arhitecturi sau sisteme de operare diferite;
- timpul necesar serializării sau deserializării unui obiect este mult mai mic decât timpul necesar salvării sau restaurării unui obiect pe baza valorilor datelor sale membre (de exemplu, în momentul deserializării unui obiect nu se apelează constructorul clasei respective);
- cea mai simplă și mai rapidă metodă de clonare a unui obiect (*deep copy*) o reprezintă serializare/deserializarea sa într-un/dintr-un tablou de octeți.

Obiectele unei clase sunt serializabile dacă respectiva clasă implementează interfața `Serializable`. Această interfață este una de marcaj, care nu conține nicio metodă abstractă, deci, prin implementarea sa clasa respectivă doar anunță mașina virtuală Java faptul că dorește să-i fie asigurat mecanismul de serializare. O clasă nu este implicit serializabilă, deoarece clasa `java.lang.Object` nu implementează interfața `Serializable`. Totuși, anumite clase standard, cum ar fi clasa `String`, clasele înfășurătoare (wrapper), clasa `Arrays` etc., implementează interfața `Serializable`.

Pentru prezentarea mecanismului de serializare/deserializare, vom considera definită clasa `Student`, care implementează interfața `Serializable`, având datele membre `String` nume, `int` grupa, un tablou `note` cu elemente de tip `int` pentru a reține notele unui student, `double` medie și o dată membră statică `facultate` de tip `String`, respectiv metodele de tip `set/get` corespunzătoare, metoda `toString()` și constructori:

```
public class Student implements Serializable
{
    private static String facultate;
    private String nume;
    private int grupa, note[];
    private double medie;
    ...
}
```

Serializarea unui obiect se realizează astfel:

- se deschide un flux binar de ieșire utilizând clasa `java.io.ObjectOutputStream`:
  ```
  FileOutputStream file = new FileOutputStream("studenti.bin");
  ObjectOutputStream fout = new ObjectOutputStream(file);
  ```

- se salvează/scrie obiectul în fișier apelând metoda `void writeObject(Object ob)`:
  ```
  Student s = new Student("Ion Popescu", 241, new int[]{10, 9, 10, 7, 8});
  fout.writeObject(s);
  ```

Deserializarea unui obiect se realizează astfel:

- se deschide un flux binar de intrare utilizând clasa `java.io.ObjectInputStream`:
  ```
  FileInputStream file = new FileInputStream("studenti.bin");
  ObjectInputStream fin = new ObjectInputStream(file);
  ```

- se citește/restaurează obiectul din fișier apelând metoda `Object readObject()`:
  ```
  Student s = (Student)fin.readObject();
  ```

Mecanismul de serializare a unui obiect presupune salvarea, în format binar, a următoarelor informații despre acesta:

- denumirea clasei de apartenență;
- versiunea clasei de apartenență, implicit aceasta fiind hash-code-ul acesteia, calculat de către mașina virtuală Java;
- valorile datelor membre de instanță.
- antetele metodelor membre.

Observații:

- Implicit NU se serializează datele membre statice și nici corpurile metodelor, ci doar antetele lor.
- Explicit NU se serializează datele membre marcate prin modificatorul transient (de exemplu, s-ar putea să nu dorim salvarea anumitor informații confidențiale: salariul unei persoane, parola unui utilizator etc.).
- Serializarea nu tine cont de specificatorii de acces, deci se vor serializa și datele/metodele private!
- În momentul sterilizării unui obiect se va serializa întregul graf de dependențe asociat obiectului respectiv, adică obiectul respectiv și toate obiectele referite direct sau indirect de el.

Exemplu:
Considerăm clasa `Nod` care modelează un nod al unei liste simplu înlănțuite:

```
class Nod implements Serializable
{
    Object data;
    Nod next;

    public Nod(Object data)
    {
        this.data = data;
        this.next = null;
    }
}
```

Folosind clasa `Nod`, vom construi o listă circulară, formată din numerele naturale cuprinse între 1 și 10, pe care apoi o vom salva/serializa în fișierul binar `lista.ser`, scriind în fișier doar primul său nod (obiectul `prim`) – restul nodurilor listei vor fi salvate/serializate automat, deoarece ele formează graful de dependențe asociat obiectului `prim`:

```
public class Serializare_listă_circulară
{
    public static void main(String[] args)
    {
        Nod prim = null, ultim = null;

        for (int i = 1; i <= 10; i++)
        {
            Nod aux = new Nod(i);

            if (prim == null) prim = ultim = aux;
            else
            {
                ultim.next = aux;
                ultim = aux;
            }
        }
        ultim.next = prim;
```

```
        System.out.println("Lista care va fi serializată:");
        Nod aux = prim;
        do
        {
            System.out.print(aux.data + " ");
            aux = aux.next;
        }
        while(aux != prim);

        try (ObjectOutputStream fout = new ObjectOutputStream(
                                        new FileOutputStream("lista.ser")))
        {
            fout.writeObject(prim);
        }
        catch (IOException ex) { System.out.println("Excepție: " + ex); }
    }
}
```

Pentru a restaura lista circulară salvată/serializată în fișierul binar `lista.ser`, vom citi/deserializa din fișier doar primul său nod (obiectul `prim`), iar restul nodurilor listei vor fi restaurate/deserializate automat, deoarece ele formează graful de dependențe asociat obiectului `prim` (evident, clasa `Nod` trebuie să fie vizibilă):

```
public class Deserializare_listă_ciculară
{
    public static void main(String[] args)
    {
        try (ObjectInputStream fin = new ObjectInputStream(
                                        new FileInputStream("lista.ser")))

        {
            Nod prim = (Nod) fin.readObject();



            System.out.println("Lista deserializata:");
            Nod aux = prim;
            do
            {
                System.out.print(aux.data + " ");
                aux = aux.next;
            }
            while(aux != prim);

            System.out.println();
        }
        catch (Exception ex)
        {
            System.out.println("Excepție: " + ex);
        }
    }
}
```

În exemplul prezentat, graful de dependențe asociat obiectului `prim` este unul ciclic, deoarece lista este circulară (deci există o referință indirectă a obiectului `prim` către el însuși), dar mecanismul de serializare gestionează fără probleme o astfel de situație complicată!

- Dacă un obiect care trebuie serializat conține referințe către obiecte neserializabile, atunci va fi generată o excepție de tipul `NotSerializableException`.
- Dacă o clasă serializabilă extinde o clasă neserializabilă, atunci datele membre accesibile ale superclasei nu vor fi serializate. În acest caz, superclasa trebuie să conțină un constructor fără argumente pentru a inițializa în procesul de restaurare a obiectului datele membre moștenite.
- Dacă se modifică structura clasei aflată pe mașina virtuală care realizează serializarea obiectelor (de exemplu, prin adăugarea unui câmp nou privat, care, oricum, nu va fi accesibil), fără a se realiza aceeași modificare și pe mașina virtuală destinație, atunci procesul de deserializare va lansa la executare excepția `InvalidClassException`. Excepția apare deoarece cele două clase nu mai au aceeași versiune. Practic, versiunea unei clase se calculează în mod implicit de către mașina virtuală Java printr-un algoritm de hash care este foarte sensibil la orice modificare a clasei. În practică, sunt diferite situații în care se dorește modificarea clasei pe mașina virtuală care realizează procesul de serializare (de exemplu, adăugând date sau metode private care vor fi utilizate doar intern), fără a afecta, însă, procesul de deserializare. O soluție în acest sens o constituie introducerea unei noi date membre în clasă, `private static final long serialVersionUID`, prin care se definește explicit versiunea clasei - caz în care mașina virtuală Java nu va mai calcula, implicit, versiunea clasei respective pe baza structurii sale. Un exemplu bun în acest sens se găsește în pagina https://www.baeldung.com/java-serial-version-uid.

EXTERNALIZAREA OBIECTELOR

Deși mecanismul de serializare este unul foarte puternic și util, totuși, el are și câteva dezavantaje:

- serializarea unui obiect nu ține cont de modificatorii de acces ai datelor membre, deci vor fi salvate în format binar și datele de tip `protected/private`, ceea ce permite reconstituirea valorilor lor prin tehnici de *reverse engineering* (https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/Chapters/3_5-Serialization.pdf);
- serializarea nu salvează datele membre statice;
- serializarea unui obiect necesită destul de mult spațiu de stocare, deoarece se salvează multe informații auxiliare (de exemplu, cele referitoare la superclasele clasei corespunzătoare obiectului);
- serializarea unui obiect se realizează destul de lent, fiind un proces recursiv vizavi de superclasă și/sau referințe către alte obiecte.

Dezavantajele menționate anterior pot fi ameliorate sau eliminate folosind mecanismul de *externalizare*, care este, de fapt, o serializare complet controlată de către programator (implicit se salvează doar numele clasei). Astfel, programatorul poate decide datele care vor fi salvate în format binar, precum și modalitatea utilizată pentru salvarea lor. De exemplu, pentru a asigura confidențialitatea unei date membre a unui obiect la serializare/deserializare (de exemplu, salariul unui angajat), este necesară criptarea/decriptarea întregului obiect, ceea ce va crește considerabil durata celor două procese. Folosind mecanismul de externalizare, se poate cripta/decripta doar data membră respectivă!

Externalizarea obiectelor unei clase este condiționată de implementarea interfeței `Externalizable`:

```
public interface Externalizable extends Serializable
{
      public void writeExternal(ObjectOutput out) throws IOException;
      public void readExternal(ObjectInput in) throws IOException,
                                                        ClassNotFoundException;
}
```

Practic, în cadrul celor două metode `writeExternal` și `readExternal`, programatorul își poate implementa proprii algoritmi de salvare și restaurare a unui obiect.

Exemplu:
Vom prezenta modul în care mecanismul de externalizare poate fi aplicat în cazul clasei `Student`, menționată anterior:

```
public class Student implements Externalizable
{
    public static String facultate;

    String nume;
    int grupa, note[];
    double medie;
    ..............................


    //se va salva denumirea facultății (prin serializare nu ar fi fost salvată,
    //deoarece este o dată membră statică)
    //nu se vor salva notele unui student
    //media va fi "criptată" folosind formula 2*medie+3
    @Override
    public void writeExternal(ObjectOutput out) throws IOException
    {
           out.writeUTF(facultate);
           out.writeUTF(nume);
           out.writeInt(grupa);
           out.writeDouble(2*medie+3);
           out.writeObject(note);
    }

    //media va fi "decriptată" folosind formula inversă (medie-3)/2
    @Override
    public void readExternal(ObjectInput in) throws IOException,
                                                      ClassNotFoundException
    {
           facultate = in.readUTF();
           nume = in.readUTF();
           grupa = in.readInt();
           medie = in.readDouble();
           medie = (medie - 3)/2;
    }
}
```

După implementarea celor două metode `writeExternal` și `readExternal`, salvarea/restaurarea datelor se va realiza la fel ca și în cazul serializării, apelând metodele `writeObject` și `readObject`.

Exemplu:
Considerăm un tablou `s` cu elemente de tip `Student`:

```
Student s[] = new Student[5];
Student.facultate =  "Facultatea de Matematica si Informatica";

s[0] = new Student(...);
.......................
```

Salvarea tabloului `s` într-un fișier binar `studenti_extern.ser` se poate realiza astfel:

```
try(ObjectOutputStream fout = new ObjectOutputStream(
                                  new FileOutputStream("studenti_extern.ser")))
{
     fout.writeObject(s);
}
catch (IOException ex)
{
     System.out.println("Excepție: " + ex);
}
```

Restaurarea tabloului `s` din fișierul binar `studenti_extern.ser` se poate realiza astfel:

```
Student s[];

try(ObjectInputStream fin = new ObjectInputStream(
                                  new FileInputStream("studenti_extern.ser")))
{
     s = (Student [])fin.readObject();
}
catch (Exception ex)
{
     System.out.println(ex);
}
```

Un aspect pe care nu trebuie să-l pierdem din vedere în momentul utilizării externalizării este faptul că se vor pierde toate facilitățile puse la dispoziție de mecanismul de serializare (salvarea automată a informațiilor despre superclasele clasei respective, salvarea automată a grafului de dependențe etc.), deci un programator va trebui să le implementeze explicit!