Java serijalizacija

Rad sa fajl sistemom

Za rad sa fajl sistemom se koristi klasa *File* iz paketa *java.io*. U klasu su enkapsulirani podaci koji opisuju jednu datoteku ili jedan direktorijum. Instanciranje klase ide ovako:

```
File fabs = new File("c:\\test.txt");
```

Parametar konstruktora može biti apsolutna ili relativna putanja. Ako se na navede početni deo (na windowsu c:/) podrazumeva se relativna putanja u odnosu na direktorijum u kome se izvršava java program. To je u *eclipse* okruženju *root* direktorijum projekta.

Sledi i primer instanciranja datoteka sa relativnim putanjama:

```
File frel = new File("images\\test.jpg");
System.out.println(frel.getAbsolutePath()); //Ispisuje se puna putanja.
System.out.println(frel.getPath()); //Ispisuje se putanja koja je uneta kao parameter konstruktora
```

Datoteka se može instancirati relativno u odnosu na *parent* direktorijum gde se nalazi koristeći konstruktor sa dva parametra:

```
File dir = new File("c:\\testdir");
File file = new File(dir, "test.txt");
```

Listanje sadržaja direktorijuma se radi preko metode *listFiles()* koja vraća opet listu fajlova.

```
File src = new File("src");
File[] files = src.listFiles();
for(File f : files) {
	System.out.println(f + " " + f.isDirectory() + " " + f.isFile());
}
```

Klasa *File* je čisto opisna klasa, to znači da se može instancirati na osnovu neke putanje koja ne mora postojati u fajl sistemu. Provera da li datoteka ili direktorijum zaista postoji u fajl sistemu se može izvršiti preko metode *exists()* koja vraća *boolean*. Ako se želi napraviti nova datoteka potrebno je pozvati metodu *createNewFile()* koja vraća *boolean* vrednost o uspešnosti operacije (ako datoteka već postoji, ništa se ne radi i vraća se *false*).

Brisanje datoteke se može odraditi preko metode *delete()*. Ako je u pitanju brisanje direktorijuma, on mora biti prazan, inače će operacija biti neuspešna. Metoda može baciti i *exception* ako java proces nema ovlašćenja pristupa datoj datoteci ili direktorijumu.

Rad sa tokovima

Prenos podataka na odredište se radi sa konceptom *Stream* u javi. Odredište može biti datoteka, odredište preko mreže, pa čak i mesto u operativnoj memoriji. Bez obzira na odredište slanje podataka se radi na isti način.

Tokovi mogu biti dolazni i odlazni.

Za dolazne tokove se koristi klasa ***InputStream*** ili neka od potklasa za specifičnu namenu. Dolazni tok predstavlja prijem podataka iz nekog izvora (čitanje podataka). Za odlazne tokove se koristi klasa ***OutputStream*** ili neka od potklasa za specifičnu namenu. Služi za slanje podataka na odredište.

Klase su apstraktne i jako jednostavne i operišu na nivou bajtova (čitanje i pisanje niza bajtova u tok). Za konkretnu namenu koriste se potklase ovih klasa. Rad sa tokovima zahtevaju da se oni na kraju zatvore ako se više ne koriste kako bi se eventualno zauzeti resursi oslobodili. Zatvaranje tokova se postiže metodom *close()*. Potrebno je pozvati unutar *finally* bloka, kako bi bili osigurani da će se uvek izvršiti. Metoda *close()* takođe može baciti *exception IOException* kao i većina drugih metoda za rad sa tokovima. U tom slučaju nemate mogućnost zagarantovanog zatvaranja toka.

Takođe postoji metoda *flush()* koja šalje sve zaostale bajtove u tok ili čita iz njega, a nalaze se u nekom baferu. Pročitati dokumentaciju za konkretni tok, da li se prilikom zatvaranja toka radi automatski *flush* podataka. Inače se može desiti da je tok ugašen, a svi podaci nisu poslati. Većina tokova radi automatski flush prilikom zatvaranja.

FileInputStream i FileOutputStream

Klase koje predstavljaju odredište i izvor koji je lociran u fajl sistemu kao standardna datoteka. Primer instanciranja za slanje podataka na odredište.

```
File f = new File("stream.txt");

FileOutputStream fos = new FileOutputStream(f);
```

Metodom *write(byte[])* se može izvršiti upis određeni niz bajtova u tok. Sledi instanciranje toka iz koga se čitaju podaci.

```
FileInputStream fis = new FileInputStream(f);
```

```
byte[] b = new byte[16];

int read = 0;
while ((read = fis.read(b)) != -1) {
      …
}
```

Standardni metod čitanja iz ulaznog toka jeste da se pročitani podaci smeste u neki privremeni bafer (bajt niz od 16 bajtova kao u primeru) metodom *read(byte[])* koja vraća koliko se bajtova zaista pročitalo.

Detaljnije o zatvaranju tokova

Tokovi se mogu zatvoriti na dva načina. Preko *try finally* bloka:

```
// Pisanje u tok.
FileOutputStream fos = new FileOutputStream(f);
```

```
try {
	byte[] b = "Ovo je neki tekst koji će se upisati u datoteku!!!".getBytes("UTF-8");
	fos.write(b);
} finally {
	fos.close();
}
```

Mana ovog pristupa je što ako imate više resursa koje trebate zatvoriti, *try finally* se drastično komplikuje:

```
FileOutputStream fos1 = new FileOutputStream(new File("dat1.txt"));
try {
	FileOutputStream fos2 = new FileOutputStream(new File("dat2.txt"));
	try {
		FileOutputStream fos3 = new FileOutputStream(new File("dat3.txt"));
		try {
			//Uzajmna obrada iz razlicitih izvora podataka...
		} finally {
			fos3.close();
		}
	} finally {
		fos2.close();
	}
} finally {
	fos1.close();
}
```

Drugi način zatvaranja tokova je preko specijalne konsrukcije *try(…)* koja je uvedena od Java 1.7.

```
try(FileOutputStream fos1 = new FileOutputStream(new File("dat1.txt"));
    FileOutputStream fos3 = new FileOutputStream(new File("dat3.txt"));
    FileOutputStream fos2 = new FileOutputStream(new File("dat2.txt"));) {
      //Uzajmna obrada iz razlicitih izvora podataka...
}
```

Između standardnih zagrada se mogu naći bilo koji objekti klasa koja implementiraju *AutoClosable* interfejs kako bi signalizirale okruženju da se mogu automatski zatvoriti. Java će automatski pozvati *close()* metodu definisanu u tom interfejsu za sve deklarisane resurse bez obzira kakav se izuzetak desio. Ovo se ne odnosi samo na *stream* objekte, već i na ostale resurse kao što je na primer konekcija ka bazi podataka, mrežni soketi i slično. Jedina mana ovog pristupa je što u trenutku pisanja ovog uputstva i dalje su dosta zastupljene ranije verzije jave (1.6, 1.5, …).

ByteArrayInputStream i ByteArrayOutputStream

Klase koje služe za upis i čitanje podataka iz operativne memorije. Mogu biti korisne za skladištenje niza bajtova za koji se u trenutku ne zna tačna njegova dužina. Sledi primer upisa stringa u operativnu memoriju kao niz bajtova.

```
// Upis bajtova u operativnu memoriju.
ByteArrayOutputStream bos = new ByteArrayOutputStream();
try {
	byte[] b = "Ovo je neki tekst koji će se upisati u operativnu memoriju kao niz
bajtova".getBytes("UTF-8");
	bos.write(b);
} finally {
```

```
        bos.close(); // Za byte array output stream close ne radi nista.
}
```

Ako se žele pročitati podaci standardno preko toka iz operativne memorije potrebno je instancirati *ByteArrayInputStream*. Konstruktor prima niz bajtova koji će predstavljati izvor iz koga se čita. U ovom slučaju niz bajtova se vadi iz izlaznog toka *ByteArrayOutputStream* preko metode *toByteArray()*. Metoda vraća ceo niz bajtova koji se upisivao preko ovog toka.

```
ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
```

Standardnom metodom *read(byte[])* se može učitati celokupni niz bajtova preko toka *ByteArrayInputStream*.

FilterInputStream i FilterOutputStream

Rad na nivou bajtova obično zahteva mukotrpan posao transformisanja u konkretne podatke i obrnuto. Suštinski svaki tok na svom najnižem nivou operiše na nivou bajtova, ali je takođe moguće podići nivo preko klasa *FilterInputStream* i *FilterOutputStream* odnosno njihovih potklasa. Ove klase služe za transformaciju podataka pre nego što pošalju u odlazni tok nižeg nivoa i obrnuto transformiranje bajtova u podatke iz dolaznog toka nižeg nivoa. Pošto se filter tokovi oslanjaju na tokove nižeg nivoa, potrebno je pri njihovom instanciranju zadati tok nižeg nivoa. Taj tok nižeg nivoa opet može biti filter tok... Ovo je postignuto preko dizajn šablona *Decorator.* Pogledati *http://www.oodesign.com/decorator-pattern.html* i identifikavati odgovarajuće klase tokova koje učestvuju u ovom šablonu.

DataInputStream i DataOutputStream

Prvi primer filter tokova jeste par *DataInputStream* i *DataOutputStream* koji služe za upis i čitanje podataka: primitivnih tipova i *String* objekata. Neki od primera metoda za tu svrhu iz klase *DataOutputStream* su: *writeChar(char)*, *writeInt(int), writeUTF(String), ...* Slične su metode i za čitanje zadatih tipova podataka preko klase *DataInputStream*. Sve te metode konvertuju podatke u niz bajtova i šalju u odlazni dok ili transformišu niz bajtova u odgovarajući podatak iz dolaznog toka.

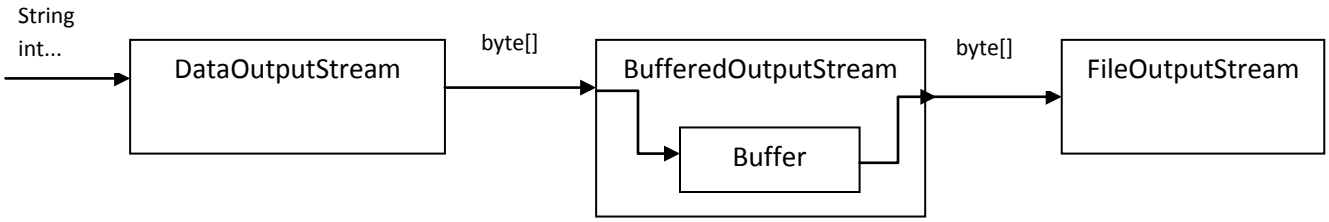
```
File f = new File("datastream.txt");
DataOutputStream dos = new DataOutputStream(new FileOutputStream(f));
try {
        dos.writeInt(10);;
        dos.writeBoolean(false);
        dos.writeUTF("String upisan preko DataOutputStream klase");
} finally {
        dos.close(); //Zatvara i tok nizeg nivoa.
}

DataInputStream dis = new DataInputStream(new FileInputStream(f));
try {
        int v = dis.readInt();
        boolean b = dis.readBoolean();
        String s = dis.readUTF();
        System.out.println(v + " - " + b + " - " + s);
} finally {
        dis.close(); //Zatvara i tok nizeg nivoa.
}
```

BufferedInputStream i BufferedOutputStream

Ukoliko postoji situacija da se vrši upis blok po blok podataka u neki tok i postoji veliki broj takvih malih blokova može doći do usporenja. Primer je datotečni tok gde pristup hard disku zahteva određeni vremenski interval. Ukoliko imamo ciklus od 100000 pristupa i u svakom koraku upisujemo 128 bajtova podataka ne iskorišćavamo pun potencijal

```
File f = new File("bufferstream.txt");
DataOutputStream dos = new DataOutputStream(new BufferedOutputStream(new FileOutputStream(f)));
try {
	writeLarge(dos);
} finally {
	dos.close();
}
```

U ovom primeru se i dalje koristi kao najviši nivo *DataOutputStream*, ali sada je niži nivo opet filter tok *BufferedOutputStream* koji primljene podatke iz višeg nivoa neće automatski slati u svoj tok, već će skladištiti u bafer operativne memorije i kada se nakupi dovoljno podataka u baferu onda će odjednom poslati podatke u tok nižeg nivoa. Na taj način se redukuje broj slanja blokova podataka u odredišni tok. U primeru slanja *int* vrednosti 100000 puta preko metode *writeInt(int)* ne koristeći bafer vreme trajanja je 676 ms. Koristeći bafer vreme se smanjuje na 8 ms (ovo je reprezentativni primer).



DeflaterOutputStream i InflaterInputStream

Kompresija podataka se takođe može transparentno odraditi koristeći filter tokove za kompresiju. Za slanje kompresovanih podataka na odredište koristi se klasa *DeflaterOutputStream*, dok se čitanje iz izvora i dekompresija vrši preko *InflaterInputStream*.

```
// Sa kompresijom deflate format.
File f = new File("deflaterstream.txt");
DataOutputStream dos = new DataOutputStream(
	new BufferedOutputStream(new DeflaterOutputStream(new FileOutputStream(f))));
try {
	writeLargeData(dos);
} finally {
	dos.close();
}

// Citanje sa dekompresijom.
DataInputStream dis = new DataInputStream(
	new BufferedInputStream(new InflaterInputStream(new FileInputStream(f))));
try {
	System.out.println(dis.readInt() + " " + dis.readInt() + " " + dis.readInt());
} finally {
	dis.close();
}
```

Ovde je opet iskorišćen *DataOutputStream* na najvišem nivou kako bi se lakše upisale vrednosti tipa int, zatim dalje se ti podaci baferuju pa tek onda kompresuju i tako kompresovani šalju na odredišni tok koji je u ovom slučaju datotečni tok.

Da bi se podaci učitali iz datoteke koja je kompresovana, potrebno je iskoristiti *InflaterInputStream* koji će odraditi dekompresiju podataka.

Takođe postoje klase *InflaterOutpuStream* za dekompresiju podataka i slanje takvih na odredište. Kao i klasa *DeflaterInputStream* koja vrši kompresiju učitanih podataka.

PrintStream

PrintStream klasa je zgodna filter tok klasa koja služi za štampanje formatiranih vrednosti kao što su stringovi i brojevi u odredišni tok. Poseduje metode kao što su *print(String), println(String)* za štampanje teksta i teksta u novoj liniji. Čuveni *System.out* objekat je baš objekat ove klase.

```
// Ispisivanje u konzoli ali kompresovano.
PrintStream ps = new PrintStream(new DeflaterOutputStream(System.out));
System.setOut(ps);

System.out.println("Ovo je neki tekst, ali se neće baš tako ispisati u konzoli!");
```

PrintStream se instancira kao i svaki drugi filter tok prosleđujući tok nižeg nivoa. U ovom slučaju se instancira tok koji se oslanja na standardni izlaz ali pre toga se kompresuju podaci pre nego što se pošalju na izlaz.

ObjectOutputStream i ObjectInputStream

Za upis kompleksnijih objekata u datoteku i čitanje iz nje je mukotrpno koristeći klasu *DataOutputStream* jer se mora voditi računa o redosledu pozivanja metoda (*writeInt, writeUTF, writeBoolean, …*), a i u tom slučaju je potrebno implementirati mehanizam konvertovanja Vaših objekata u ove primitivne podatke. Za ovu svrhu se koriste objektni tokovi *ObjectOutputStream* za upis kompleksnijih objekata u tok i *ObjectInputStream* za čitanje podataka koji će se automatski konvertovati u te iste objekte.

ObjectOutputStream poseduje glavnu metodu *writeObject(Object)* koja bukvalno serijalizuje objekat u niz bajtova i šalje u tok. Takođe *ObjectInputStream* ima metodu *readObject()* koja vraća objekat koji je upisan prethodno u tok.

Ne može se svaki objekat serijalizovati ovim mehanizmom. Da bi određeni objekat mogao da se serijalizuje mora da njegova klasa implementira interfejs *Serializable* koji je prazan i služi samo za indikaciju da je klasa serijalizabilna. Ako se pokuša serijalizovati objekat klase koja ne implementira *Serializable* baciće se *exception java.io.NotSerializableException*.

```
public class Automobil implements Serializable {
	private String model;
	private int godište;
	private int brojVrata;
	private Date datumProizvodnje;
	private Motor motor;
```

```
public class Motor implements Serializable {
```

Takođe sve vrednosti atributa koje klasa poseduje direktni i indirektni (nasleđeni od roditeljske klase) moraju da podrže serijalizaciju, odnosno i njihove klase moraju biti serijalizabilne. Sve standardne klase u Javi za koje mislite da bi mogle da se serijalizuju zaista mogu. Slede neki primeri standardnih Javinih klasa i tipova koji se mogu serijalizovati:

- Primitivni tipovi (*boolean*, *int*, *double*, …)
- Enumeracije (*enum*)
- *java.lang.String*
- *java.util.Date*
- *Wrapper* klase za primitivne tipove (*Integer, Double, Long, BigDecimal*, ...)
- Nizovi
- Većina kolekcija: *java.util.ArrayList, java.util.HashMap, java.util.HashSet, java.util.Vector*, …
- Pa čak i elementi GUI-a (*Swing*): *JFrame, JButton*, …
- …

Neki primeri klasa koje se ne mogu serijalizovati:

- *java.awt.Toolkit, javax.swing.SwingUtilities, java.lang.Thread*, …

Proces serijalizacije odnosno slanja i čitanja iz toka ide na standardni način:

```
Motor bmwMotor = new Motor(1.998, 222, "sdgsdkjgkl", Tip.DIZEL);
Automobil bmw320 = new Automobil("320E", 2005, 5, new Date(), bmwMotor);
Automobil opel = new Automobil("Astra", 2006, 5, new Date(), bmwMotor);
Automobil[] automobili = new Automobil[] { bmw320, opel };

File f = new File("objectstream.txt");
ObjectOutputStream oos = new ObjectOutputStream(new BufferedOutputStream(new
FileOutputStream(f)));
try {
	oos.writeObject(automobili);
} finally {
	oos.close(); //Zatvara i tok nizeg nivoa.
}

ObjectInputStream ois = new ObjectInputStream(new BufferedInputStream(new FileInputStream(f)));
try {
	Automobil[] ucitaniAutomobili = (Automobil[])ois.readObject();
} finally {
	ois.close();
}
```

Prilikom konstrukcije *ObjectOutputStream* objekta potrebno je proslediti odredišni tok u koga će se stvarno poslati niz serijalizovanih bajtova. U ovom slučaju odredišni tok je baferovan datotečni tok. U ovom primeru se serijalizuje niz automobila. Prilikom čitanja radi se sličan posao. Rezultat je novi niz automobila učitan iz datoteke koji nemaju apsolutno nikakve veze po referencama sa objektima koji su prethodno bili serijalizovani u tok. Sa druge strane serijalizovani objekti kada se deserijalizuju iz toka imaće istu uvezanost. U gornjem primeru oba učitana automobila iz datoteke imaće istu referencu na motor. Dakle, serijalizacija vodi računa o međusobnim referencama.

Ukoliko ima potrebe da se određeni atributi klase ne uključe u proces serijalizacije, iz razloga na primer jer se ne mogu serijalizovati, onda je potrebno deklarisati takav atribut sa ključnom reči *transient*.

```
public class Automobil implements Serializable {
```

```
…
	private Motor motor;

	private transient Vlasnik vlasnik;
```

U ovom primeru *Automobil* ima vlasnika, objekat klase *Vlasnik* koja nije serijalizabilna klasa. Tako da prilikom serijalizacije automobila kada se prolazi kroz sve attribute i vrši serlijalizacija istih u niz bajtova, naleteće se i na atribut vlasnik i ako on nije *null* baciće se *exception*. Rešenje je staviti ključnu reč *transient*. To znači da će se zadati atribut preskočiti prilikom serijalizacije. Kada se kompletni objekat učitava nazad u memoriju iz toka, tranzijentni atributi će imati *default* vrednost, a to je *null* za objekte, 0 za *int*, *false* za *boolean*, itd.

Serial Version ID

Možda ste primetili da u *Eclipse* okruženju kada se klasa deklariše da bude serijalizabilna automatski ima *warning* da nije definisan statički atribut *serialVersionUID*. Ovaj statički atribut koji je u stvari i konstanta definiše na neki način verziju klase. Preporučljivo je izgenerisati *random* vrednost u svakoj klasi koja je serijalizabilna.

```
public class Automobil implements Serializable {

	private static final long serialVersionUID = 7961803530405751475L;
```

Ako klasa ne poseduje ovu statičku konstantu dešava se sledeće. Napravili ste v1.0 vaše aplikacije i isporučili klijentu da je koristi. Prilikom korišćenja klijent snima objekte automobil u odredišnu datoteku. Zatim ste isporučili v2.0 klijentu u kome klasa ima recimo dodatni atribut koji nije ranije postojao ili je neki postojeći izbrisan. Klijent pokušava da učita datoteku iz ranije verzije. Desiće se izuzetak jer deskriptor klase više nije isti. Drugim rečima pokušavate učitati objekat iz datoteke, a njegova klasa se izmenila. Ako v1.0 klase ima definisanu konstantu *serialVersionUID* i v2.0 ima istu tu konstantu i istu vrednost onda će se pokušati odraditi automatsko učitavanje. Ako neki atribut ne postoji u v2.0 ignorisaće se njegova vrednost iz datoteke i obrnuto, ako je neki uveden novi biće postavljen na *default* vrednost jer ne postoji u v1.0 datoteke. Ako je datoteka toliko izmenjena da nije više podržano učitavanje iz ranijih verzija, promeniti vrednost konstante *serialVersionUID* u novijoj verziji klase kako bi izvršili signalizaciju okruženju da izbaci izuzetak prilikom učitavanja objekta koji se odnosio na stariju verziju klase.

Rukovanje sa procesom seijalizacije

U svaku klasu koja je serijalizabilna moguće je umetnuti metode sa određenom deklaracijom kao podrška upravljanju serijalizacijom koje će se automatski pozivati od strane okruženja prilikom serijalizacije svakog objekta.

```
public class Automobil implements Serializable {

	…

	private void writeObject(java.io.ObjectOutputStream out) throws IOException {
		// Neki dodatni posao upisa u datoteku.
		Date now = new Date();
		out.writeObject(now);

		// Default serijalizacija this objekta
		out.defaultWriteObject();

	}

	private void readObject(java.io.ObjectInputStream in) throws IOException,
ClassNotFoundException {
```

```
            Date d = (Date) in.readObject();
            System.out.println("Vreme upisa datoteke je bilo: " + d);

            in.defaultReadObject();
      }

}
```

U ovom primeru metode *writeObject* i *readObject* moraju biti deklarisane baš ovako kako je napisano. Njih će pozvati okruženje prilikom serijalizacije objekta ove klase *Automobil*. Ukoliko želite da ubacite neki dodatni posao, a da se i dalje oslonite na standardni mehanizam serijalizacije potrebno je pozvati metodu *defaultWriteObject()* u slučaju upisa objekta u tok. Metoda će umesto vas odraditi dosadni deo serijalizacije svih atributa i atributa natklase. U ovom primeru se jednostavno upisuje u datoteku na početku vreme serijalizacije ovog objekta. Prilikom čitanja objekta iz toka, pozvaće se metoda *readObject* ako je definisana. Ovde se jednostavno čita prvi objekat iz toka koji je tipa datum i ispisuje se u konzoli vreme koje je stajalo u datoteci. Zatim se oslanja na standardni način deserijalizacije pozivom metode *defaultReadObject* koja će popuniti ovaj objekat sa podacima iz datoteke.

Postoji i treća metoda za rukovanje deserijalizacijom

```
private void readObjectNoData() throws ObjectStreamException;
```

Poziva se u slučaju da je klasa promenila natklasu, a pokušava se učitati objekat ranije verzije klase koja nije imala tu natklasu, pa je u tom slučaju potrebno izvršiti inicijalizaciju stanja natklase.

Pored ove tri metode postoji i set metoda *writeReplace* i *readResolve* koje možete definisati u svakoj klasi koja je serijalizabilna:

```
public class Automobil implements Serializable {
…
      Private Object writeReplace() throws ObjectStreamException {
            //Mozete zameniti da se ne serijalizuje ovaj objekat nego neki drugi
            //Ili mozda objekat potklase ili malo izmenjeni objekat
            return this;
      }

      private Object readResolve() throws ObjectStreamException {
            this.vlasnik = new Vlasnik("Petar", "Petrović");
            //Mozete vratiti bilo sta i objekat bilo koje klase.
            //I to ce predstavljati krajnji rezultat citanja objekta iz datoteke.
            return this;
      }
```

Prednost *readResolve* metode je što možete izvršiti potrebnu inicijalizaciju tranzijentnih objekata da ne bi bili *null* prilikom završetka učitavanja objekta iz datoteke. Jer ti atributi mogu biti *private* i nemati *setter* pa ne bi bilo ni moguće izvršiti inicijalizaciju na drugi način.

Sumarno, proces serijalizacije ide ovako kada se pozove metoda *writeObject(Object)* nad klasom *ObjectOutputStream*:

- Ako postoji metoda *writeReplace* pozvaće se i vraćeni objekat će biti predmet serijalizacije.
- Ako postoji metoda *write* pozvaće se i preusmeriće se na nju kompletan proces serijalizacije objekta u niz bajtova.

Sumarno, proces deserijalizacije ide ovako kada se pozove metoda *readObject()* nad klasom *ObjectInputStream*:

- Ako postoji metoda *read* pozvaće se i preusmeriće se na nju kompletan proces deserijalizacije objekta na osnovu niza bajtova iz toka
- Ako postoji metoda *readResolve* pozvaće se i rezultat te metode će biti vraćeni objekat metodi *readObject()*.

Upotreba serijalizacije za kloniranje objekata

Objekti koji su serijalizabilni ne moraju da se serijalizuju u datotečni tok. Mogu da se serijalizuju i u bafer u memoriji pa da se na osnovu tog izvora deserijalizuju nazad kao objekti i to potpuno novi (klonirani). Drugim rečima umesto *FileOutputStream* upotrebiti klasu *ByteArrayOutputStream*.

```
ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(bos);
try {
	oos.writeObject(frame);
} finally {
	oos.close();
}

ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
ObjectInputStream ois = new ObjectInputStream(bis);
try {
	JFrame newFrame = (JFrame) ois.readObject();
	newFrame.setVisible(true);
} finally {
	ois.close();
}
```

Napomena: prilikom učitavanja objekta iz toka ne poziva se nijedan konstruktor njegove klase. Nego se rekonstrukcija objekta radi interno u okviru Java virtuelne mašine.

Rad sa čitačima i pisačima (*Reader i Writer*)

Apstraktne klase *Reader i Writer* služe za upis i čitanje niza karaktera odnosno stringa po zadatom *encoding*-u. Ovde se tok podataka ne smatra niz bajtova nego niz karaktera (char). Klase su apstraktne, tako da treba koristiti odgovarajuću potklasu za konkretnu namenu.

BufferedWriter i BufferedReader

Konkretne klase koje služe za skladištenje u bafer i kada se on napuni šalje se na odredište, odnosno vraća kao rezultat čitanja iz izvora.

```
File f = new File("writer.txt");
// Klasa OutputStreamWriter je most izmedju karakter toka i byte toka.
BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(f)));
try {
	writer.write("Neki tekst koji će biti snimljen u datoteku");
} finally {
	writer.close();
}

BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(f)));
try {
	String s = reader.readLine();
	System.out.println(s);
} finally {
	reader.close();
}
```

Čitači i pisači se takođe moraju zatvarati preko metode *close()* koja automatski i radi *flush* podataka. Bafer čitač može primiti kao parametar konstruktora drugi čitač. Isto važi i za pisače. Ako se želi osloniti na konkretan tok podataka (*byte* tok) koristi se most *OutputStreamWriter* i *InputStreamReader* između čitača i tokova kao u gornjem primeru.

PrintWriter

Za pisanje stringova u više linija postoji zgodna klasa *PrintWriter*.

```
File f = new File("printwriter.txt");
PrintWriter pw = new PrintWriter(new BufferedWriter(new OutputStreamWriter(new
DeflaterOutputStream(new FileOutputStream(f)))));
try {
	pw.println("Ovo je neki tekst!!!");
	pw.println("U novoj liniji.");
	pw.println();
	pw.println("Opet neki tekst");
} finally {
	pw.close();
}

BufferedReader reader = new BufferedReader(new InputStreamReader(new InflaterInputStream(new
FileInputStream(f))));
try {
	String line = null;
	while((line = reader.readLine()) != null) {
		System.out.println(line);
	}
} finally {
	reader.close();
}
```

Xstream biblioteka

Eksterna biblioteka koja olakšava proces serijalizacije Javinih objekata u XML ili JSON format.

http://x-stream.github.io/download.html

Sa sajta u sekciji *Download* je moguće skinuti najnoviju verziju biblioteke. Uključiti glavni *jar* u *Build path* i sve zavisne *jar* datoteke iz foldera *xstream* koje se mogu naći u zipovanom preuzetom fajlu.

```
Motor bmwMotor = new Motor(1.998, 222, "sdgsdkjgkl", Tip.DIZEL);
Vlasnik vlasnik = new Vlasnik("Ivan", "Ivanović");
Automobil bmw320 = new Automobil("320E", 2005, 5, new Date(), bmwMotor, vlasnik);
Automobil opel = new Automobil("Astra", 2006, 5, new Date(), bmwMotor, vlasnik);
Automobil[] automobili = new Automobil[] { bmw320, opel };

File f = new File("xstream.xml");
OutputStream os = new BufferedOutputStream(new FileOutputStream(f));
try {
	XStream xs = new XStream();
	// Naziv tag-a u xml umesto punog naziva klase.
	xs.alias("auto", Automobil.class);
	xs.alias("motor", Motor.class);
	// Serijalizacija u xml.
	String s = xs.toXML(automobili); // Kao string.
	xs.toXML(automobili, os); // U tok.
	System.out.println(s);
	// Deserijalizacija iz xml-a.
	Automobil[] ucitaniAutomobili = (Automobil[]) xs.fromXML(s);
	System.out.println(ucitaniAutomobili[0]);
	System.out.println(ucitaniAutomobili[1]);

} finally {
	os.close();
}
```

XStream se može konfigurisati prilično. Moguće je definisati koji atributi da uđu u serijalizaciju, koji ne preko metode *omitField*. Moguće je definisati nazive tagova u odredišnom xml-u koristeći metodu *alias*. Prednost ove biblioteke je što klase ne moraju uopšte da implementiraju *Serializable interfejs*. Ako implementiraju, biblioteka će uzeti u obzir ključne reči *transient* i dopunske metode (*readResolve, ...*) za podršku upravljanju serijalizacijom.

Metodom *toXml* i *fromXml* se pokreće serijalizacija i deserijalizacija respektivno.

Takođe je moguće serijalizovati u format JSON koristeći odgovaorajući *driver* prilikom instanciranja *Xstream* objekta.

```
// Poseban drajver treba proslediti u konstruktoru za JSON.
XStream xs = new XStream(new JettisonMappedXmlDriver());
```

Ostalo se radi sasvim identično kao i u slučaju serijalizacije XML-a.