Mrežno računarstvo

Java

Tokovi podataka

Tokovi (streams)

- Ogroman deo onoga što rade mrežni programi je prosto ulaz i izlaz: premeštanje bajtova od jednog sistema do drugog
- Čitanje podataka koje nam šalje server ne razlikuje se od čitanja fajla
- Slanje teksta klijentu ne razlikuje se od pisanja u fajl

Tokovi

- I/O u Javi je izgrađen na tokovima.
- Ulazni tok čita podatke, izlazni ih piše
- Razne klase kao što su *java.io.FileInputStream*,... čitaju i pišu određene izvore podataka
- Svi izlazni tokovi imaju iste osnovne metode za pisanje podataka, takođe i ulazni za njihovo čitanje
- Nakon kreiranja toka, možemo ignorisati detalje o tome šta se tačno čita ili piše

Filter tokovi

- Filter tokovi mogu se olančati bilo na ulazni bilo na izlazni tok. Filteri mogu podatke dok ih čitaju ili pišu – npr. kriptovati ili ih kompresovati – ili obezbediti dodatne metode za konvertovanje podataka u druge formate.
- Npr. klasa *java.io.DataOutputStream* ima metod koji konvertuje *int* u 4 bajta i piše ih u izlazni tok

Čitači i pisači

- Čitači i pisači mogu se olančati na ulazne ili izlazne tokove da bi omogućili programima da čitaju i pišu tekst (karaktere) umesto bajtova. Čitači i pisači mogu rukovati raznovrsnim kodiranjima teksta, uključujući višebajtne karakterske skupove kao što je npr. UTF-8

Tokovi

- Tokovi su sinhroni: kada program (zapravo nit) traži da tok čita ili piše podatke, on čeka da se podaci pročitaju ili ispišu pre nego što može da radi nešto drugo.
- Java podržava neblokirajući I/O koristeći kanale i bafere. Neblokirajući I/O je nešto komplikovaniji, ali mnogo brži za web servere.
- Uobičajeno, osnovni tokovi su sve što je potrebno koristiti za klijente. Kanali i baferi zavise od tokova.

Izlazni tokovi (Output streams)

- Glavna izlazna klasa je:

  java.io.OutputStream

  public abstract class OutputStream
- Osnovni metodi za pisanje podataka
  - public abstract void write(int b) throws IOException
  - public void write(byte[] data) throws IOException
  - public void write(byte[] data, int offset, int length) throws IOException
  - public void flush() throws IOException
  - public void close() throws IOException

Izlazni tokovi

- Potklase od *OutputStream* koriste ove metode za pisanje podataka na određeni medij.
- Npr. *FileOutputStream* koristi ove metode za pisanje u fajl
- *TelnetOutputStream* za pisanje u mrežnu konekciju
- *ByteArrayOutputStream* u proširivi niz bajtova
- Ali, gde god da se pišu podaci, uglavnom se koriste ovih istih 5 metoda
- Ponekad čak i ne znamo tačno koje vrste je tok u koji pišemo podatke

Izlazni tokovi

- *TelnetOutputStream* (stari *sun* paketi)
- Vraćaju ga razni metodi raznih klasa iz paketa *java.net*, poput metoda *getOutputStream()* klase *java.net.Socket*
- Ipak, ovi metodi su deklarisani tako da vraćaju *OutputStream*, a ne konkretno njegovu potklasu *TelnetOutputStream*. (polimorfizam: ako umete da koristite superklasu, znaćete i iz nje izvedene klase)

write(int b)

- Osnovni metod klase *OutputStream* je write(int b). Uzima *int* od 0 do 255 kao argument i piše odgovarajući bajt u izlazni tok.
- Metod je deklarisan kao apstraktan jer potklase moraju da ga prilagode tako da rukuje odgovarajućim medijem.
- Npr. *ByteArrayOutputStream* može implementirati ovaj metod čistim Java kodom koji kopira bajt u niz. Međutim, *FileOutputStream* će morati da koristi native kod koji ”zna” kako da piše fajlove na host-platformu

write()

- Iako metod **write**() prima *int* kao argument, on zapravo piše *unsigned byte*. Java nema *unsigned byte* kao tip, pa se koristi *int*. Jedina stvarna razlika između *unsigned byte* i *signed byte* je u interpretaciji. Oba se sastoje od 8 bitova i kada se piše *int* u mrežnu konekciju koristeći **write**(int b), samo 8 bitova se smešta na žicu. Ako je *int* izvan opsega 0-255 prosleđen kao argument, bajt najmanje težine se piše, a ostala 3 bajta se ignorišu (efekat kastovanja *int*-a u *byte*).
- U retkim situacijama može se desiti da klasa izbacuje *IllegalArgumentException* ili uvek ispisuje 255, pa je bolje ne računati na ovakvo ponašanje, ako je moguće

Protokol generisanja karaktera

- Npr. protokol generisanja karaktera definiše server koji odašilje ASCII tekst.
- Najpopularnija varijanta ovog protokola šalje 72-karakterske linije koje sadrže štampajuće ASCII karaktere (između 33 i 126 uključujući, izuzev belina i kontrolnih karaktera)
- Prva linija sadrži karaktere 33 do 104, uređene
- Druga linija 34 do 105, treća 35 do 106 i ovo se nastavlja do linije 23 koja sadrži karaktere 55 do 126
- Dalje, linija 24 sadrži karaktere 56 do 126, za kojima ponovo ide karakter 33
- Linije se završavaju sa carriage return (ASCII 13) i linefeed (ASCII 10)

Protokol generisanja karaktera

- Kako je ASCII 7-bitni, svaki karakter se šalje kao pojedinačni bajt
- Dakle, protokol je moguće implementirati pravolinijski korišćenjem osnovnih write() metoda
- **PRIMER 1**, metod generateCharacters()
- Serverska klasa za generisanje karaktera prosleđuje *OutputStream out* metodu **generateCharacters**()
- Bajtovi se pišu u *out* jedan po jedan.
- Najveći deo aritmetike se sastoji u tome da se u petlji karakteri rotiraju u odgovarajućem opsegu

Protokol generisanja karaktera

- Čitav metod je deklarisan tako da izbacuje *IOException*. To je bitno jer server za generisanje karaktera se završava samo kada klijent zatvori konekciju. Java kôd vidi to kao *IOException*.

Protokol generisanja karaktera

- Pisanje jednog po jednog bajta je često neefikasno. Npr. svaki TCP segment koji ode na Ethernet kartu sadrži bar 40 dodatnih bajtova neophodnih za rutiranje i korekciju grešaka.
- Ako se svaki bajt šalje pojedinačno, možemo puniti mrežu sa 41 puta više podataka nego što mislimo da ih je.

Baferisanje

- Zato, većina TCP/IP implementacija vrši baferisanje podataka
- To znači da akumulira bajtove u memoriji i šalje ih na njihovo eventualno odredište samo kada se akumulira određeni broj ili protekne određeno vreme
- Međutim, ako imamo spremno više od jednog bajta, nije loša ideja poslati ih sve odjednom
- Korišćenje **write**(byte[] data) ili **write**(byte[] data, int offset, int length) je obično mnogo brže nego pisanje svakog elementa niza data pojedinačno.
- **PRIMER 1**, metod generateCharacters1()
- implementacija metoda generateCharacters1() šalje liniju po liniju tako što smešta celu liniju u niz bajtova

Baferisanje

- Algoritam za računanje koji bajt se kada ispisuje je isti kao u prethodnoj implementaciji
- Glavna razlika je u tome što se bajtovi pakuju u niz pre nego što se pišu u mrežu. Takođe, rezultujući *int* mora se kastovati u *byte* pre nego što se smesti u niz.
- To malopre nije bilo potrebno jer je metod **write**() deklarisan tako da prima *int*

Baferisanje

- Tokovi mogu biti baferisani u softveru, direktno u Java kodu, kao i u mrežnom hardveru.
- Tipično, to se postiže olančavanjem *BufferedOutputStream* ili *BufferedWriter* na tok
- Zato, kada se završi sa pisanjem podataka, bitno je uraditi ”flush” izlaznog toka
- Metod *flush*() forsira baferisani tok da pošalje podatke čak i kada bafer nije pun

Baferisanje, flush()

- Bitno je raditi *flush* toka bez obzira da li Vi mislite da je to potrebno ili ne.
- Tok *System.out* je baferisan želeli mi to ili ne
- Ako flushing nije neophodan za neki tok, to je operacija niske cene. Ako je neophodan, vrlo je neophodan! Njegov izostanak vodi problemu koji je teško dijagnostifikovati
- Zato, trebalo bi raditi flush svih tokova neposredno pre njihovog zatvaranja. Inače, moguće je da dođe do gubljenja podataka koji su se zatekli u baferu prilikom zatvaranja toka.

close()

- Po završetku rada sa tokom, on se zatvara *close*() metodom
- On oslobađa resurse pridružene toku, poput fajl deskriptora ili portova.
- Nakon zatvaranja izlaznog toka, dalji pokušaji pisanja u njega dovode do izbacivanja *IOException*
- Međutim, neke vrste tokova dopuštaju da se nakon toga obavljaju neke operacije nad objektom. Npr. zatvoreni *ByteArrayOutputStream* može biti konvertovan u stvarni niz bajtova

Ulazni tokovi

- Osnovna ulazna Java klasa je:

  java.io.InputStream

  public abstract class InputStream
- Osnovni metodi za čitanje neobrađenih bajtova
  - public abstract int read() throws IOException
  - public int read(byte [] input) throws IOException
  - public int read(byte [] input, int offset, int length) throws IOException
  - public long skip(long n) throws IOException
  - public int available() throws IOException
  - public void close() throws IOException

Ulazni tokovi

- Konkretne potklase koriste ove metode za čitanje podataka sa određenog medija
- Npr. *FileInputStream* čita podatke iz fajla
- *TelnetInputStream* čita iz mrežne konekcije
- *ByteArrayInputStream* čita iz niza bajtova
- Odakle god da se čita, uglavnom se koristi samo ovih 6 metoda.
- Ponekad nije tačno poznato iz koje vrste toka se čita

Ulazni tokovi

- *TelnetInputStream* je nedokumentovana klasa skrivena unutar nekadašnjeg *sun.net* paketa
- Instance se mogu dobiti raznim metodima klasa iz java.net paketa:
- npr. *openStream*() metod klase *java.net.URL*
- ovi metodi su deklarisani tako da vraćaju tip *InputStream*. Opet polimorfizam: instanca potklase se može koristiti kao instanca superklase

read()

- Osnovni metod klase *InputStream* je **read**() (bez argumenata)
- Metod čita jedan bajt sa ulaznog toka i vraća ga kao *int* od 0 do 255. Kraj toka se signalizira vraćanjem vrednosti -1.
- Metod čeka i blokira izvršavanje koda koji sledi dok ne dobije spreman bajt za čitanje.
- Ulaz i izlaz mogu biti spori, pa ako program radi još nešto od značaja, pokušati da se I/O stavi u sopstvenu nit.

read()

- metod je deklarisan kao apstraktan jer potklase moraju da ga prilagode svojim medijima.
- npr. *ByteArrayInputStream* može implementirati ovaj metod čistim Java kodom koji kopira bajt iz svog niza
- Međutim, *TelnetInputStream* mora koristiti native biblioteku koja ”zna” kako da čita podatke iz mrežnog interfejsa na host-platformi

Primer

- fragment čita 10 bajtova iz *InputStream*-a *in* i smešta ih u niz bajtova *input*. Ako se dođe do kraja toka, petlja se završava ranije
- byte[] input = new byte [10];

```
for(int i=0; i < input.length; i++){
    int b = in.read();
    if(b == -1) break;
    input[i] = (byte)b;
}
```

Primer

- Iako **read**() čita samo bajt, vraća *int*. Tako, neophodno je kastovanje pre smeštanja rezultata u niz bajtova.
- Ovo proizvodi označeni bajt od -128 do 127 umesto neoznačenog od 0 do 255 kojeg vraća metod **read**(). Međutim, sve dok nam je jasno sa čim radimo, to ne predstavlja veliki problem
- Konverzija označenog u neoznačeni bajt moguća je sa:

```
int i = b >= 0 ? b : 256 + b
```

Baferisanje

- Čitanje pojedinačnog bajta je neefikasno kao i njegovo pisanje
- Zato postoje dva predefinisana read() metoda koja pune zadati niz bajtovima pročitanim iz toka: **read**(byte[] input) i **read**(byte[] input, int offset, int length)
- Prvi pokušava da da napuni zadati niz *input*
- Drugi pokušava da napuni zadati podniz od *input*, počev od pozicije *offset* i nastavljajući *length* bajtova

read()

- Metodi pokušavaju da napune niz, ne uspevaju nužno.
- Pokušaj može da ne uspe iz nekoliko razloga. Npr. neki od traženih bajtova se mogu pročitati, ali ne svi. Pokušavanje čitanja 1024 bajta, a 512 je stvarno stiglo sa servera, ostali su još u putu. Oni će na kraju stići, ali trenutno nisu raspoloživi
- Ove 2 verzije **read**() metoda vraćaju broj stvarno pročitanih bajtova

primer

- byte[] input = new byte [1024];
  int bytesRead = in.read(input);
- ovaj fragment koda pokušava da pročita 1024 bajta iz *InputStream*-a *in* u niz *input*.
- Međutim, ako je samo 512 bajtova dostupno, to je sve što će biti pročitano, i *bytesRead* će biti postavljeno na 512
- **Da bismo se osigurali da će svi bajtovi koje želimo zaista biti pročitani, smestimo read() u petlju koja uzastopno čita sve dok se niz ne popuni.**

Primer

```
int bytesRead = 0;
int bytesToRead = 1024;
byte[] input = new byte[bytesToRead];
while(bytesRead < bytesToRead)
   bytesRead += in.read(input, bytesRead,
bytesToRead – bytesRead);
```

- Ova tehnika je posebno važna za mrežne tokove. Šanse su da ako je fajl dostupan u potpunosti, svi bajtovi fajla su takođe dostupni.

- Međutim, pošto je mreža znatno sporija od CPU, vrlo lako se može desiti da program prazni mrežni bafer pre nego što svi podaci stignu.

read()

- Sva 3 **read**() metoda vraćaju -1 kao signal kraja toka. Ako se tok završi a ima još podataka koji nisu pročitani, višebajtni **read**() vraća podatke dok se bafer ne isprazni. Sledeći poziv bilo kog **read**() metoda vraća -1.
- -1 se nikada ne smešta u niz
- Niz uvek sadrži samo stvarne podatke
- Prethodni kod ne valja jer ne razmatra mogućnost da svih 1024 bajtova ne stignu nikad
- Popravljanje te greške zahteva testiranje povratne vrednosti metoda **read**() pre njenog dodavanja na *bytesRead*.

Primer

```
int bytesRead = 0;
int bytesToRead = 1024;
byte[] input = new byte [bytesToRead];
while(bytesRead < bytesToRead){
   int result = in.read(input, bytesRead,
        bytesToRead – bytesRead);
   if( result == - 1) break;
   bytesRead += result;
}
```

available()

- Ako ne želite da čekate dok svi bajtovi koje želite ne postanu momentalno dostupni, možete koristiti metod **available**() za utvrđivanje koliko bajtova može biti pročitano bez blokiranja.
- Metod vraća minimalan broj bajtova koje možete pročitati.
- Možda ćete moći da pročitate i više, ali moći ćete bar toliko koliko **available**() predlaže.

Primer

- int bytesAvailable = in.available();
  byte[] input = new byte[bytesAvaliable];
  int bytesRead = in.read(input, 0, bytesAvailable);
  // continue with rest of program immediately...
- U ovom slučaju, možemo tvrditi da je *bytesRead* tačno jednako *bytesAvailable*.
- Ne možemo, međutim tvrditi da je *bytesRead* veće od 0. Moguće je da nije bilo dostupnih bajtova.
- Na kraju toka, **available**() vraća 0
- Generalno, **read**(byte[] input, int offset, int length) vraća -1 na kraju toka, ali ako je *length* 0, on ne primećuje kraj toka i vraća 0.

skip()

- U retkim prilikama želimo da preskočimo neke podatke bez čitanja
- Manje je od koristi pri čitanju mrežnih konekcija, nego fajlova
- Mrežne konekcije su sekvencijalne i uobičajeno prilično spore, tako da nije značajno vremenski zahtevnije pročitati podatke od njihovog preskakanja
- Fajlovi imaju slučajan pristup, tako da se preskakanje jednostavno implementira repozicioniranjem pokazivača fajla

close()

- Kao i kod izlaznih tokova, nakon što program završi sa ulaznim tokom, trebalo bi da ga zatvori pozivom **close**() metoda
- Ovim se oslobađaju resursi pridruženi toku, poput fajl deskriptora i portova.
- Nakon zatvaranja ulaznog toka, dalji pokušaji čitanja iz njega uzrokuju izbacivanje *IOException*.
- Međutim, neke vrste tokova mogu dozvoljavati neke operacije nad objektom.

Označavanje i resetovanje

- Klasa *InputStream* ima i 3 manje korišćena metoda koja omogućavaju programima back up i ponovno čitanje već pročitanih podataka:
  - *public void mark(int readAheadLimit)*
  - *public void reset() throws IOExcept*
  - *public boolean markSupported()*
- U cilju ponovnog čitanja podataka, tekuća pozicija toka označi se metodom **mark**(). Kasnije, moguće je resetovati tok na markiranu poziciju korišćenjem metoda **reset**().
- Uzastopna čitanja onda vraćaju podatke počev od označene pozicije.
- Međutim, možda nije moguće resetovati proizvoljno daleko unazad. Broj bajtova koje je moguće pročitati od oznake i resetovati određen je argumentom metoda **mark**(). Ako se pokuša sa predalekim resetovanjem, izbacuje se *IOException*
- U svakom trenutku u toku može postojati samo jedna oznaka. Označavanje druge lokacije briše prethodnu oznaku.
- Ovo ne podržavaju svi ulazni tokovi
- Može se proveriti metodom **markSupported**() (vraća *true* ako je podržano)
- Ako označavanje i resetovanje nije podržano, **mark**() ne radi ništa, a **reset**() izbacuje *IOException*.
- Jedine dve klase ulaznih tokova u java.io koje uvek podržavaju su *BufferedInputStream* i *ByteArrayInputStream*
- *TelnetInputStream* može podržavati ako se najpre olanča na baferisani ulazni tok

Filter tokovi

- *InputStream* i *OutputStream* su prilično sirove klase. One čitaju bajtove pojedinačno ili u grupama, i to je sve
- Odluka o tome šta predstavljaju ti bitovi – cele brojeve, IEEE754 brojeve u pokretnom zarezu ili Unicode tekst – u potpunosti je prepuštena programeru i kodu.
- Međutim, postoje određeni veoma uobičajeni formati podataka: 32-bitni big-endian integer-i, 7-bitni ASCII, 8-bitni Latin1 ili višebajtni UTF-8 tekstovi, .zip fajlovi
- Java obezbeđuje veliki broj filter klasa koje se mogu pridodati sirovim tokovima u cilju prevođenja sirovih bajtova u i iz ovih i drugih formata

Filteri

- Filteri postoje u 2 varijante: filter tokovi i čitači i pisači.
- Filter tokovi primarno rade sa sirovim podacima kao bajtovima: kompresovanjem ili interpretiranjem podataka kao binarnih brojeva
- Čitači i pisači rukuju specijalnim slučajem teksta različito kodiranog (UTF-8 ili ISO-8859-1 npr.)

Filteri

- Filter tokovi su smešteni povrh sirovih tokova poput *TelnetInputStream* ili *FileOutputStream*
- Čitači i pisači su smešteni povrh sirovih tokova, filter tokova ili drugih čitača i pisača
- Filter tokovi, međutim, ne mogu biti povrh čitača ili pisača.

Filteri

- Filteri su organizovani u lanac. Svaki link lanca prihvata podatke od prethodnog filtera ili toka i prosleđuje podatke sledećem linku lanca.
- ... slika i primer (kompresovani, kriptovani tekstualni fajl)
- Svaki filter izlazni tok ima iste **write**(), **close**() i **flush**() metode kao i *java.io.OutputStream*
- Svaki filter ulazni tok ima iste **read**(), **close**() i **available**() metode kao i *java.io.InputStream*
- *BufferedInputStream* i *BufferedOutputStream* i imaju samo te metode
- U većini slučajeva, filter tok ima i *public* metode sa dodatnom svrhom (npr. **unread**() u *PushbackInputStream*)

Olančavanje filtera

- Filteri se povezuju sa tokovima korišćenjem svojih konstruktora
- Sledeći kod baferiše ulaz iz fajla *data.txt*
- Najpre se kreira *FileInputStream* objekat *fin* prosleđivanjem imena fajla kao argumenta odgovarajućem konstruktoru. Zatim se kreira *BufferedInputStream* objekat *bin* prosleđivanjem *fin* kao argumenta odgovarajućem konstruktoru
- FileInputStream fin = new
  FileInputStream(”data.txt”);
  BufferedInputStream bin = new
  BufferedInputStream(fin);
- Od ovog trenutka moguće je koristiti **read**() metode za oba *bin* i *fin* za čitanje podataka iz fajla *data.txt*. Međutim, mešanje poziva različitih tokova povezanih sa istim izvorom nije dobro. Veći deo vremena, treba koristiti poslednji filter u lancu za stvarno čitanje i pisanje.
- Jedan način da se ne izazove ovakva greška jeste da se namerno izgubi referenca na ”donji” ulazni tok:
- InputStream in = new FileInputStream(”data.txt”);
  in = new BufferedInputStream(in);

VEŽBA: *kopiranje fajla (ne mora biti tekstualni) sa jedne lokacije na drugu*

Ulančavanje filtera

- ili

```
DataOutputStream dout =
    new DataOutputStream(
      new BufferedOutputStream(
        new FileOutputStream(”data.txt”)
      )
    );
```

Konekcija je trajna. Filteri se ne mogu ”raskačiti” od toka.

Čitati i pisati samo u poslednji filter u lancu!

- Postoje situacije kada je neophodno koristiti metode većeg broja filtera iz lanca.
- Npr. ako čitamo Unicode tekstualni fajl, možda želimo da pročitamo byte order mark u prva 3 bajta kako bismo odredili da li je fajl kodiran kao big-endian UCS-2, little-endian UCS2 ili UTF-8 i zatim selektovali odgovarajući Reader filter.
- Ili, ako smo konektovani na web server, možda želimo da pročitamo zaglavlje koje server šalje, kako bismo našli Content-encoding i koristili to kodiranje za Reader filter kojim čitamo telo severovog odgovora
- U tim slučajevima neophodno je sačuvati i koristiti reference na svaki od pridruženih tokova, međutm **ni pod kojim okolnostima ne treba čitati niti pisati u filtere koji nisu poslednji u lancu.**

Čitači i pisači

- Mnogi programeri imaju lošu naviku pisanja koda kao da je sav tekst ASCII ili bar u native kodiranju platforme.
- To nije tačno za HTTP i mnoge druge protokole.
- Javin native karakterski skup je UTF-16
- Kada kodiranje nije ASCII, pretpostavka da su bajtovi isto što i karakteri više ne stoji
- Zato, Java obezbeđuje skoro kompletnu hijerarhiju klasa ulaznih i izlaznih tokova za rad sa karakterima umesto sa bajtovima

Čitači i pisači

- 2 apstraktne superklase definišu osnovni API za čitanje i pisanje karaktera
- *java.io.Reader* određuje API za čitanje karaktera, a *java.io.Writer* za pisanje
- Gde god ulazni i izlazni tokovi koriste bajtove, čitači i pisači koriste Unicode karaktere.
- Filter čitači i pisači mogu biti dodati na druge čitače i pisače da obezbede dodatne servise ili interfejse
- Najvažnije potklase su *InputStreamReader* i *OutputStreamWriter*.

- *InputStreamReader* čita bajtove iz (underlying) ulaznog toka i prevodi ih u Unicode karaktere u skladu sa zadatim kodiranjem
- *OutputStreamWriter* prima Unicode karaktere od programa koji se izvršava, a zatim ih prevodi u bajtove koristeći zadato kodiranje i piše bajtove u (underlying) izlazni tok.

- Paket *java.io* takođe obezbeđuje nekoliko sirovih čitač i pisač klasa koje čitaju karaktere bez da zahtevaju direktni ulazni tok:
- *FileReader* *FileWriter*
- *StringReader* *StringWriter*
- *CharArrayReader* *CharArrayWriter*
- Nisu od velikog značaja za mrežno programiranje (prve dve rade sa fajlovima, a preostale 4 unutar Jave)

Pisači

- Klasa *Writer – java.io.OutputStream*
- apstraktna je i ima dva *protected* konstruktora
- kao i *OutputStream* nikada se ne koristi direktno, već polimorfno preko jedne od svojih potklasa
- ima 5 metoda **write**(), **flush**() i **close**()

Writer

- protected Writer()
- protected Writer(Object lock)
- public abstract void write(char[] text, int offset, int length) throws IOException
- public void write(int c) throws IOException
- public void write(char[] text) throws IOException
- public void write(String s) throws IOException
- public void write(String s, int offset, int length) throws IOException
- public abstract void flush() throws IOException
- public abstract void close() throws IOException

Primeri

- char[] network ={’N’, ’e’, ’t’, ’w’, ’o’, ’r’, ’k’};
  w.write(network, 0, network.length);
- w.write(network);
- for(int i=0; i<network.length; i++)
  w.write(network[i]);
- w.write(”Network”);
- w.write(”Network”, 0, 7);

- Sve su ovo bili različiti načini za postizanje istog cilja
- Koliko i kojih bajtova će biti zapisano zavisi od kodiranja koje *w* koristi
- Ako koristi big-endian UTF-16 (14 bajtova), little-endian UTF-16 (drugih 14 bajtova – onih istih samo razmenjeni par-nepar). Latin1 ili UTF-8 (7 bajtova)

- Pisači mogu biti baferisani, bilo direktno ulančavanjem sa *BufferedWriter* ili indirektno jer im je underlying izlazni tok baferisan
- **flush**() i **close**() – ista priča kao kod *OutputStream*

OutputStreamWriter

- najbitnija potklasa klase *Writer*
- prima karaktere iz Java programa
- konvertuje ih u bajtove u skladu sa zadatim kodiranjem
- upisuje ih u underlying izlazni tok
- Konstruktor zadaje izlazni tok i kodiranje

public OutputStreamWriter(OutputStream out, String encoding) throws UnsupportedEncodingException

public OutputStreamWriter(OutputStream out)

- valid encoding (dokumentacija za native2ascii tool)

npr. ISO8859_5, ASCII, UTF8, UTF-16, UTF_32, Cp1252

- drugi konstruktor –podrazumevano kodiranje platforme

- OutputStreamWriter w =
  new OutputStreamWriter(
  new FileOutputStream(”OdysseyB.txt”),
  ”Cp1253”);

  w.write(”αβγδεζ”);
- **PRIMER 2**, writeUTF8File
- Osim konstruktora, *OutputStreamWriter* ima samo uobičajene *Writer* metode i metod koji vraća kodiranje:
- *public String getEncoding()*

Čitači

- Klasa *Reader – java.io.InputStream*
- apstraktna, 2 *protected* konstruktora
- ni ona se nikada ne koristi direktno, već samo kroz svoje potklase
- ima 3 metoda **read**(), **skip**(), **close**(), **ready**(), **mark**(), **reset**(), **markSupported**()

Reader

- protected Reader()
- protected Reader(Object lock)
- public abstract int read(char[] text, int offset, int length) throws IOException
- public int read() throws IOException
- public int read(char[] text) throws IOException
- public long skip(long n) throws IOException
- public boolean ready()
- public boolean markSupported()
- public void mark(int readAheadLimit) throws IOException
- public void reset() throws IOException
- public abstract void close() throws IOException

- read() metod vraća jedan Unicode karakter kao int sa vrednošću od 0 do 65535 ili -1 na kraju toka
- read(char[] text) pokušava da napuni niz text karakterima i vraća stvaran broj pročitanih karaktera ili -1 na kraju toka
- read(char[] text, int offset, int length) pokušava da pročita length karaktera u podniz od text koji počinje od offset i nastavlja se length karaktera. Takođe vraća stvaran broj pročitanih karaktera ili -1 na kraju toka
- skip(long n) preskače n karaktera ... (sve ostalo analogno kao ranije, osim:)
- ready() ima istu svrhu kao available() ali ne istu semantiku – vraća boolean koji kaže da li čitač može da čita bez blokiranja. available() je vraćao minimalan broj bajtova koje je moguće pročitati bez blokiranja. Pošto ovde neka kodiranja, kao što je UTF-8 koriste različit broj bajtova za različite karaktere, teško je unapred reći koliko karaktera čeka u baferu bez njihovog čitanja iz bafera

InputStreamReader

- najznačajnija potklasa od *Reader*
- čita bajtove iz underlying ulaznog toka
- npr. iz *FileInputStream* ili *TelnetInputStream*
- konvertuje ih u karaktere u skladu sa zadatim kodiranjem i vraća ih
- Konstruktor zadaje ulazni tok i kodiranje

  public InputStreamReader(InputStream in)

  public InputStreamReader(InputStream in, String encoding) throws UnsupportedEncodingException

Primer

- metod čita ulazni tok i konvertuje ga u jedan Unicode string koristeći MacCyrilic kodiranje:
- public static String getMacCyrillicString(InputStream in)
  throws IOException{
  InputStreamReader r = new
  InputStreamReader(in, ”MacCyrillic”);
  StringBuffer sb = new StringBuffer();
  int c;
  while((c=r.read())!=-1) sb.append((char)c);
  r.close();
  return sb.toString();
  }
- **PRIMER 3**, getUTF8String

Filter čitači i pisači

- Klase *InputStreamReader* i *OutputStreamWriter* menjaju interfejs nad ulaznim i izlaznim tokom od bajt-orijentisanog u karakter-orijentisani
- Nakon toga, dodatni karakter-orijentisani filteri mogu se izgraditi nad čitačima i pisačima korišćenjem klasa *java.io.FilterReader* i *java.io.FilterWriter*

Filter čitači i pisači

- Postoji mnoštvo potklasa koje vrše specifično filtriranje:
- *BufferedReader BufferedWriter*
- *LineNumberReader*
- *PushbackReader*
- *PrintWriter*

Baferisani čitači i pisači

- Klase *BufferedReader* i *BufferedWriter* su karakter-orijentisani ekvivalenti bajt-orijentisanih klasa *BufferedInputStream* i *BufferedOutputStream*
- *BufferedInput*/*OutputStream* koriste interni niz bajtova kao bafer, dok *BufferedReader*/*Writer* koriste interni niz *char*-ova

BufferedReader i BufferedWriter

- Kada program čita iz *BufferedReader*, tekst se uzima iz bafera, umesto direktno iz underlying ulaznog toka ili drugog tekstualnog izvora.
- Kada se bafer isprazni, napuni se ponovo sa što je moguće teksta više, čak i ako nije sav momentalno neophodan, čime se ubrzavaju buduća čitanja
- Kada program piše u *BufferedWriter*, tekst se smešta u bafer. Tekst se premešta u underlying izlazni tok ili drugo odredište samo kada se bafer napuni ili kada pisač eksplicitno uradi flush, što čini pisanje mnogo bržim nego što je to inače slučaj

BufferedReader i BufferedWriter

- imaju uobičajene metode pridružene čitačima i pisačima, poput **read**(), **ready**(), **write**(), **close**()
- Obe imaju 2 konstruktora koja ulančavaju *BufferedReader* ili *BufferedWriter* na underlying čitač ili pisač i postavljaju veličinu bafera. Ako veličina nije podešena, koristi se podrazumevana veličina od 8192 karaktera
- *public BuferredReader(Reader in, int bufferSize)*
- *public BufferedReader(Reader in)*
- *public BufferedWriter(Writer out)*
- *public BufferedWriter(Writer out, int bufferSize)*

Primer

- Raniji primer **getMacCyrrilicString**() je bio neefikasan jer čita karaktere jedan po jedan. Kako je MacCyrrillic 1-bajtni karakterski skup, on takođe čita bajtove jedan po jedan
- Međutim, pravolinijski se može ubrzati ulančavanjem *BufferedReader*-a na *InputStreamReader* sa:

Primer

- public static String getMacCyrillicString(InputStream in)
  throws IOException{
  Reader r = new InputStreamReader(in,
  ”MacCyrillic”);
  r = new BufferedReader(r, 1024);
  StringBuffer sb = new StringBuffer();
  int c;
  while((c = r.read()) != -1) sb.append((char)c);
  r.close();
  return sb.toString();
  }

Primer

- Sve što je bilo potrebno da bi se ovaj metod baferizovao bila je jedna dodatna linija koda. Ništa ostalo se ne menja, jer su jedini korišćeni *InputStreamReader* metodi **read**() i **close**() deklarisani u superklasi *Reader* i nasleđeni i u *BufferedReader*
- **VEŽBA:** *kopiranje tekstualnog UTF-8 fajla sa jedne lokacije na drugu*

readLine()

- Klasa *BufferedReader* takođe poseduje metod **readLine**() koji čita jednu liniju teksta i vraća je kao *String*
- *public String readLine() throws IOException*
- Olančavanjem *BufferedReader* na *InputStreamReader* mogu se korektno čitati linije karakterskih skupova različitih od podrazumevanog kodiranja platforme

readLine() ← VIŠE NE VAŽI!!!

- problem: zaustavlja svoju nit kada su tokovi linije koje se zavrsavaju CR-ovima, što je obično slučaj sa tokovima izvedenim iz Macintosh-a ili Macintosh tekstualni fajlovi. Zato je neophodno izbegavati ovaj metod u mrežnim programima
- Međutim, nije teško napisati bezbednu verziju ove klase koja korektno implementira **readLine**() metod

BufferedWriter

- klasa *BufferedWriter* dodaje jedan novi metod, koji ne postoji u njenoj superklasi: *newLine*()
- *public void newLine() throws IOException*
- Ovaj metod ubacuje platformski-zavisan separator linija (string) na izlaz
- *line.separator* sistemsko svojstvo tačno određuje koji je to string: verovatno *linefeed* na Unix-u i Mac OS X, a *carriage return* na Mac OS 9, kao i *carriage return/line feed* na Windows-u
- **Kako mrežni protokoli generalno određuju zahtevani terminator linija, ne bi trebalo koristiti ovaj metod u mrežnom programiranju. Umesto toga, eksplicitno pisati terminator linija koji zahteva protokol.**

LineNumberReader

- potklasa od BufferedReader koja prati koji je broj tekuće linije. To se u svakom trenutku može dobiti sa
- public int getLineNumber()
- Podrazumevano, broj prve linije je 0
- Broj tekuće i svih linija koje slede moguće je promeniti metodom
- public void setLineNumber(int lineNumber)

LineNumberReader

- Ovaj metod podešava samo brojeve linija koje javlja getLineNumber(). Ne menja mesto sa kog se čita tok
- Osim uobičajenih Reader metoda, LineNumberReader ima samo još 2 konstruktora
- public LineNumberReader(Reader in)
- public LineNumberReader(Reader in, int bufferSize)
- LineNumberReader je potklasa od BufferedReader, pa ima interni karakterski bafer čija veličina se može postaviti drugim konstruktorom. Podrazumevana veličina je 8192 karaktera

PushbackReader

- PushbackReader – PushbackInputStream klasa
- glavna razlika: vraća karaktere, a ne bajtove
- ima 3 unread() metoda koja vraćaju karaktere u čitačev ulazni bafer
  - public void unread(int c) throws IOException
  - public void unread(char[] text) throws IOException
  - public void unread(char[] text, int offset, int length) throws IOException
- Prvi metod vraća pojedinačni karakter, drugi niz karaktera, treći zadati podniz počev od text[offset] i zaključno sa text[offset+length-1]
- Podrazumevano, veličina pushback bafera je 1 karakter. Tu veličinu je moguće podesiti drugim konstruktorom
- public PushbackReader(Reader in)
- public PushbackReader(Reader in, int bufferSize)
- Pokušaj vraćanja više karaktera nego što može da stane u bafer izbacuje IOException

PrintWriter

- osim konstruktora, ima gotovo identičnu kolekciju metoda kao PrintStream:
- public PrintWriter(Writer out)
- public PrintWriter(Writer out, boolean autoFlush)
- public PrintWriter(OutputStream out)
- public PrintWriter(OutputStream out, boolean autoFlush)
- public void flush()
- public void close()
- public boolean checkError()
- protected void setError()
- public void write(int c)
- public void write(char[] text, int offset, int length)
- public void write(char[] text)
- public void write(String s, int offset, int length)
- public void write(String s)
- public void print: boolean, char, int, long, float, double, char[], String, Object
- public void println()
- println: boolean, char, …, Object

PrintWriter

- Ako underlying pisač na odgovarajući način rukuje konverzijama karakterskog skupa, onda to čine i svi metodi PrintWriter klase
- Međutim, to još uvek nije dovoljno dobro za mrežno programiranje
- PrintWriter klasa ima probleme zavisnosti od platforme (println() metodi koriste platformski zavisan line separator)
- Takođe, klasa “jede” sve izuzetke. Mrežni programi moraju biti spremni da rukuju neočekivanim prekidima toka podataka, što se radi obradom izuzetaka. Međutim, PrintWriter hvata sve izuzetke koje izbacuje pridruženi izlazni tok. Zasniva se na flegu greške, a programer je dužan da ga proverava nakon svakog poziva metodom checkError(). Ukratko, notifikacija o grešci je potpuno neadekvatna za nepouzdane mrežne konekcije.

Pisanje teksta

- Zaključak: koristiti OutputStreamWriter
- pisanje linije:

  write(<sadrzaj_linije> + <oznaka_kraja_linije>)