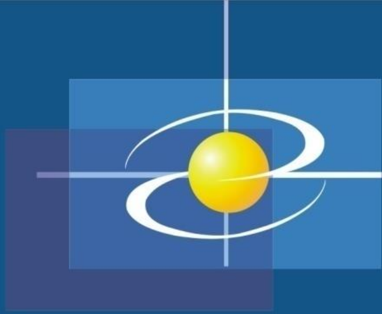
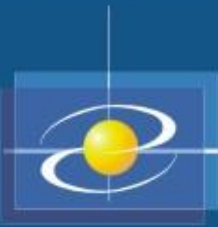


Programski jezik JAVA
PREDAVANJE 4
2020

Prezentacija kreirana na osnovu sljedeće literature :
Dejan Živković: Osnove Java programiranja; Bruce Eckel: Misliti na Javi



Izuzeci (exceptions) i ulaz/izlaz

- Greške i izuzetne situacije mogu se desiti u izvršavanju svake aplikacije. Radi se o prekidu normalnog izvršavanja programa.
- To mogu biti greške u logici programiranja, kao npr:
  - Dijeljenje sa nulom
  - Pristup elementima polja izvan specificiranih granica.
- Ovo su **izuzetne** situacije u programu koje nastaju izvršavanjem izraza koji narušavaju semantiku jezika.
- Postoje i drugi tipovi **izuzetnih situacija**, kao što su:
  - Nedostupnost sistemskog resursa poput Interneta.
  - Nepostojanje lokalne datoteke na disku.
  - Nedovoljne količine memorije.
  - Netačan unos podataka od strane korisnika.
- Aplikacije pisane u raznim programskim jezicima mogu reagovati na ovakve situacije na različite, često nepredvidljive načine.
- Mehanizam za rad u izuzetnim situacijama (**error handling**).

Izuzeci (exceptions) i ulaz/izlaz

- Primjer: Program učitava string i pretvara ga u integer:

```
import java.util.*;
public class izuzeci {
	public static void main(String[] args) {
		Scanner x = new Scanner(System.in);
		 int s = Integer.parseInt(x.nextLine());
		 System.out.println("s= " +s);
	}
   }
```

- Ako unesemo 1 program će ispisati s `= 1` na standardni izlaz (output).
- Ako unesemo **a** – imamo izuzetnu situaciju tokom izvršavanja programa. Normalan tok se prekida i umjesto ispisa `System.out.println("s= " +s);` imamo:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "a"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:449)
    at java.lang.Integer.parseInt(Integer.java:499)
    at izuzeci.main(izuzeci.java:6)
```

- radno okruženje nije reagovalo nepredvidljivo, već je u liniji koda **`int s=Integer.parseInt(x.nextLine());`** u nemogućnosti konvertovanja stringa u integer prekinulo normalno izvršavanje programa i kontrola je prenijeta na drugo mjesto izvršavnja gdje se ispisala gornja poruka.

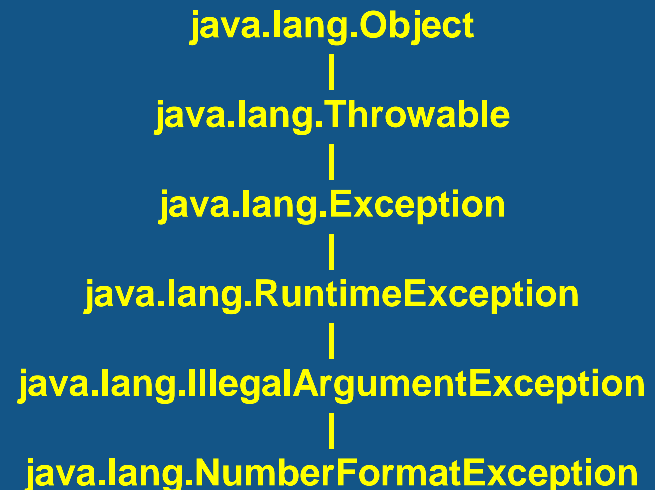
Izuzeci (exceptions) i ulaz/izlaz

- Šta se desilo? Odgovor leži u deklaraciji metode parseInt:

```
public static int parseInt(String s) throws NumberFormatException
```

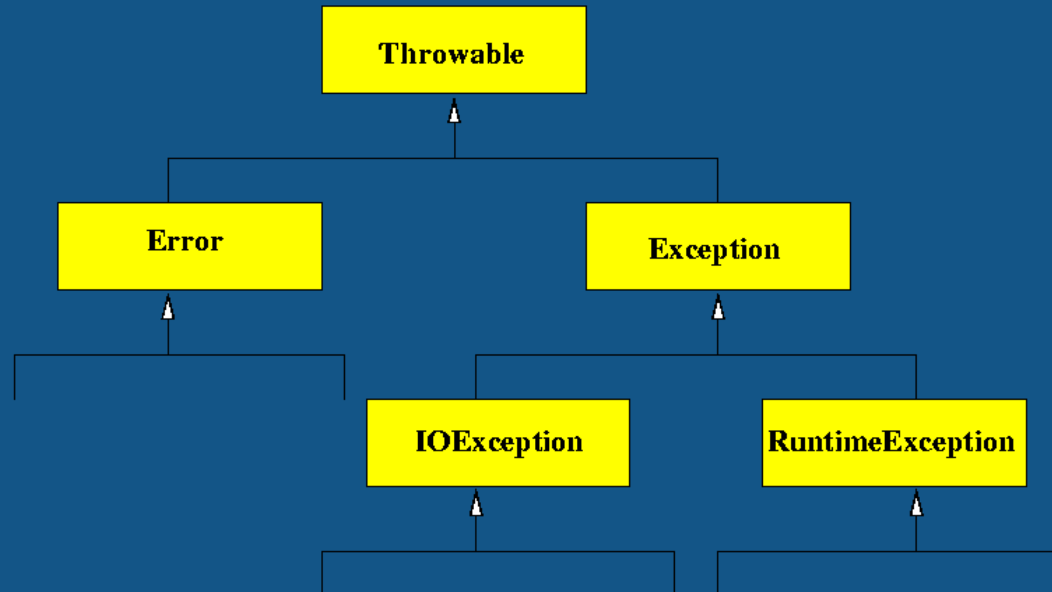
- U ovom slučaju metoda parseInt, kada naiđe na argument koji se ne da konvertovati u integer, uzrokuje da radno okruženje kreira objekt iz klase java.lang.NumberFormatException i preda ga radnom okruženju. Na taj način radno okruženje reaguje na izuzetnu situaciju.
- Da bi vidjeli šta se dalje dešava, najprije pogledajmo koje informacije sadrži takav objekt. Pogledajmo zato hijerahiju klasa u kojoj se nalazi java.lang.NumberFormatException:
- Throwable je nadklasa za sve

  greške (errors) i izuzetke (exeptions).



Podjela na greške i izuzetke

- Ovakva kontrola izuzetaka nije zadovoljavajuća, jer smo možda trebali uraditi još nešto prije završetka programa. Prema tome, moramo riješiti sljedeće:
  - Treba nam način da procesiramo izuzetak.
  - Da znamo koji izuzetak da procesiramo.
- U slučaju izuzetne situacije (greške) Java izbacuje (*throws*) objekat koji enkapsulira informaciju o grešci koja se javila.
- Svaki takav objekat (izuzetak) pripada nekoj klasi koja proširuje klasu Throwable (`java.langThrowable`)



Izuzeci tipa RunTimeException

- Za skoro sve izuzetke koji su obuhvaćeni podklasama klase Exception potrebno je uključiti kod za njihovu obradu ili program neće proći kompajliranje
- RunTimeException izuzeci se tretiraju drugačije. Prevodilac dozvoljava njihovo ignorisanje.
- Razlog: ovi izuzeci se najčešće pojavljuju kao posljedica ozbiljnijih grešaka u kodu, čija obrada ne bi mogla ništa značajno promeniti.
- Neke njene podklase:
- ArithmeticException: neispravan rezultat aritmetičke operacije poput dijeljenja sa nulom
- IndexOutOfBoundsException: indeks koji je izvan dozvoljenih granica za objekat poput niza, stringa ili vektora
- NegativeArraySizeException: upotreba negativnog indeksa niza
- NullPointerException: poziv metoda ili pristup podatku članu null objekta
- ArrayStoreException: pokušaj dodeljivana reference pogrešnog tipa elementu niza
- ClassCastException: pokušaj kastovanja objekta neodgovarajućeg tipa
- SecurityException: pokušaj narušavanja sigurnosnih pravila (security manager).

Podjela na greške i izuzetke

- Java dijeli izuzetke na provjeravane (*checked exceptions*) i neprovjeravane (*unchecked exceptions*):
  - Provjeravani izuzeci su oni za koje prevodilac provjerava da li ih program procesira. To su svi izuzeci u `java.lang.Exception` osim `java.langRunTimeException`.
  - Neprovjeravani izuzeci su oni za koje prevodilac ne provjerava da li smo ih procesirali. To su sve podklase od `java.lang.Error` i `java.langRunTime.Exception`.
- Java koristi try-catch-finally šemu za procesiranje izuzetaka.

```
try{
                /*
                  u try bloku dolazi ono što pokušavamo izvršiti
                  -ako uspijemo idemo na finally blok
                  -ako ne na catch blok
                */
          ..................
           }//kraj try bloka
           catch (NumberFormatException e) {
          ..................
            }//kraj catch bloka

            finally{
           /*
            ovaj dio nije obavezen
            -prevodioc se ne buni ako ga nema
           } //kraj finally bloka
```

Primjer

```
1  import java.util.*
2   public class NoviA {
3  public static void main(String[] args) {
4     try{
5
6       Scanner x = new Scanner(System.in);
7               int s = Integer.parseInt(x.nextLine());
8         System.out.println(“s= " +s);
9
10     }//kraj try bloka
11     catch (NumberFormatException e) {
12
13     System.out.println("Izuzetak ispisan na standardni uotput: " + e.getMessage());
14      }//kraj catch bloka
15
16     finally{
17
18       System.out.println("Ovo se uvijek izvrsi!");
19     } //kraj finally bloka
20
21          System.out.println("Program nastavi s izvrsavanjem.");
22          int s=Integer.parseInt("1234");
23          System.out.println(“s= " +s);
24     }
25  }
```

Rezultat izvršavanja

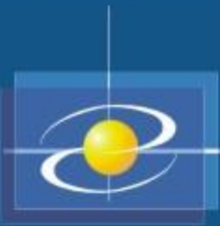
- Izvršavanjem java NoviA za unijetu vrijednost aaa imamo ovakav ispis:

```
Izuzetak ispisan na standardni uotput: For input string: "aaa"
Izuzetak ispisan na error output:
java.lang.NumberFormatException: For input string: "aaa"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:447)
    at java.lang.Integer.parseInt(Integer.java:497)
    at NoviA.main(NoviA.java:13)
Ovo se uvijek izvrsi!
Program nastavi s izvrsavanjem.
s= 1234
```

- Try blok može slijediti više catch blokova.

```
try{
...

  }//kraj try bloka
catch (NumberFormatException e) {
...
}//kraj catch bloka
catch (ArrayIndexOutOfBoundsException e) {
...
}//kraj catch bloka
catch (Exception e) {//uhvatimo sve ostale
...
}//kraj catch bloka
```



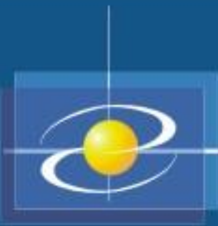
Primjer..

- Kod može eksplicitno izbaciti izuzetak pomoću `throw` naredbe.
- Ako neka metoda može izbaciti *provjeravani izuzetak*, a sama ge ne procesira, onda taj izuzetak mora biti deklarisan u samoj metodi.
- Ilustrirajmo ovo zadnje pravilo. Na primjer, konstruktor klase **java.io.FileReader** može izbaciti provjeravani izuzetak **FileNotFoundException**. Treba procesirati u catch bloku kao ovdje:

```
// Verzija s procesiranjem izuzetka

public void method()
{
        try{

            FileReader fr = new FileReader("moj.file");
            .........
        }
        catch(FileNotFoundException e){
             e.printStackTrace();
             System.exit(0);
        }
}
```



Nastavak...

- Ako ne znamo kako procesirati **FileNotFoundException** možemo ga ignorisati, ali ga moramo deklarisati na sljedeći način:

```
// Verzija bez procesiranja izuzetka

public void method() throws FileNotFoundException
{

        FileReader fr = new FileReader("moj.file");
        .........

}
```

- Sada će **FileNotFoundException** biti proslijeđen pozivnoj metodi, sve dok se ne nađe odgovarajući `catch` blok. Ako takvog nema, radno okruženje procesira izuzetak zaustavljanjem programa i ispisom sadržaja steka.

Datoteke u Javi

- U paketu **java.io** nalaze se klase za rad sa datotekama i direktorijumima. Funkcionalnost koju omogućavaju te klase sastoji se od kreiranja, brisanja i preimenovanja datoteka i direktorijuma, čitanja iz datoteke i pisanja u datoteku, itd.
- Datoteke i direktorijumi se modeluju klasom **java.io.File**. Objekti te klase mogu predstavljati i datoteke i direktorijume (između kojih Java, ne pravi velike razlike), ali u klasi **File** nema metoda za čitanje/pisanje datoteka. Ta je funkcionalnost, zbog svoje kompleksnosti, ali i univerzalnosti postupaka, modelovana u nizu drugih klasa.
- Za čitanje i pisanje koristi se koncept *stream*-a (kao u programskom jeziku C).
- *Ulazni stream* je svaki objekt iz kog se može čitati niz bajtova, dok je *izlazni stream* objekt u koji se može upisati niz bajtova. Ulazni stream se modeluje apstraktnom klasom **`java.io.InputStream`**, a izlazni klasom **`java.io.OutputStream`**.
- Na taj način proces pisanja i čitanja postaje uniforman, nezavistan od izvora iz koga čitamo ili destinacije u koju pišemo. Tako je pisanje/čitanje u lokalni sistem datoteka identično pisanju i čitanju preko mreže (komunikaciji sa Web-serverom i slično).

Nastavak...

- Proces čitanja i pisanja usložnjen je činjenicom da postoji više vrsta datoteka:
  - Binarne datoteke i tekstualne datoteke;
  - Datoteke sa sekvencijalnim pristupom i datoteke sa direktnim pristupom.
- Binarna datoteka je niz bajtova i osnovno čitanje i pisanje vrši se bajt po bajt. Klase koje nude tu funkcionalnost proširuju apstraktne klase **java.io.InputStream** i **java.io.OutputStream**.
- Tekstualne datoteke predstavljaju niz znakova i stoga su ljudima čitljive.
- Kako Java za prikaz znakova koristi **Unicode** (znak se pamti u dva bajta) to se, u principu, osnovne operacije čitanja i pisanja vrše u grupama od dva bajta.
- Klase koje omogućuju tekstualno pisanje i čitanje imaju za osnovu apstraktne klase **java.io.Reader** i **java.io.Writer**.
- Većina klasa omogućava sekvencijalni pristup podacima, dok za direktan pristup treba koristiti klasu **java.io.RandomAccessFile**.

Klasa `File`

- Objekti klase `java.io.File` reprezentuju datoteke i direktorijume. Najjednostavnija konstrukcija objekta je pomoću imena datoteke ili direktorijuma:

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

- Treba uočiti da kreiranjem objekta tipa `File` nije kreirana i datoteka, **već samo objekt** koji reprezentuje datoteku i čuva njeno ime.
- Objekt tipa `File` pamti čitavu putanju datoteke; ako konstruktoru damo samo relativno ime datoteke (bez direktorijuma u kojem se nalazi) on će uzeti da je datoteka u tekućem direktorijumu (onom u kome se izvršava program).
- Sljedeći program pokazuje da objekt tipa `File` može predstavljati nepostojeću datoteku.
- Postojanje datoteke ispitujemo pomoću metode `exists()`, dok datoteku možemo kreirati metodom `createNewFile()`.

Nastavak-program

```
1  import java.io.*;
2
3  public class TestFile {
4
5      public static void main(String [] args){
6
7          File f = new File(“fajl.txt");
8          // Put datoteke
9          System.out.println("Path = "+f.getAbsolutePath());
10         // Da li datoteka postoji?
11         System.out.println("exists() = "+f.exists());
12
13         try{
14           // Kreirajmo praznu datoteku.
15           boolean flag = f.createNewFile();
16           // Ako datoteka već postoji createNewFile() vraća false.
17           if(flag == false) System.out.println("Datoteka već postoji.");
18         }
19         // Dužni smo procesuirati
20         catch(IOException e){
21             e.printStackTrace();
22         }
23     }
24 }
```

Nastavak...

- Na različitim platformama putanje datoteka se različito prikazuju. Na primjer, separator između direktorijuma i subdirektorijuma pod Unixom/Linuxom je znak "/", dok je pod Windows-ima to znak "\".
- Da bi se izbjegla zavisnost od platformi, klasa `File` nudi statičku promjeljivu članicu:

```
static String separator
```

- koja sadrži sistemski zavisan separator. Njega bismo koristili na sljedeći način:

```
File f = new File("Documents"+File.separator+"test.txt");
```

Binarne datoteke

- Za čitanje binarnih datoteka koristimo klase koje proširuju apstraktnu klasu `java.io.InputStream`.
- Jedan dio te hijerarhije prikazan je na sljedećoj slici:

Binarno pisanje i čitanje

```
1   // Binarno pisanje i čitanje datoteka pomoću klasa
2   // DataInputStream, FileInputStream / DataOutputStream, FileOutputStream
3
4   import java.io.*;
5
6   public class TestData{
7
8   public static void main(String[] args) throws FileNotFoundException, IOException
9       {
10          boolean b = true;
11          int     i = -23456;
12          char    c = 'c';
13          String  s = "DataOutputStream";
14          double  d = 234.4e+8;
15
16          // Otvaranje/kreiranje datoteke
17          FileOutputStream fos = new FileOutputStream("io_example.dat");
18          DataOutputStream dos = new DataOutputStream(fos);
19
20          // Upisivanje pojedinih tipova podataka u datoteku
21          dos.writeBoolean(b);
22          dos.writeInt(i);
23          dos.writeChar(c);
24          dos.writeUTF(s);
25          dos.writeDouble(d);
26          dos.close();
```

Nastavak...

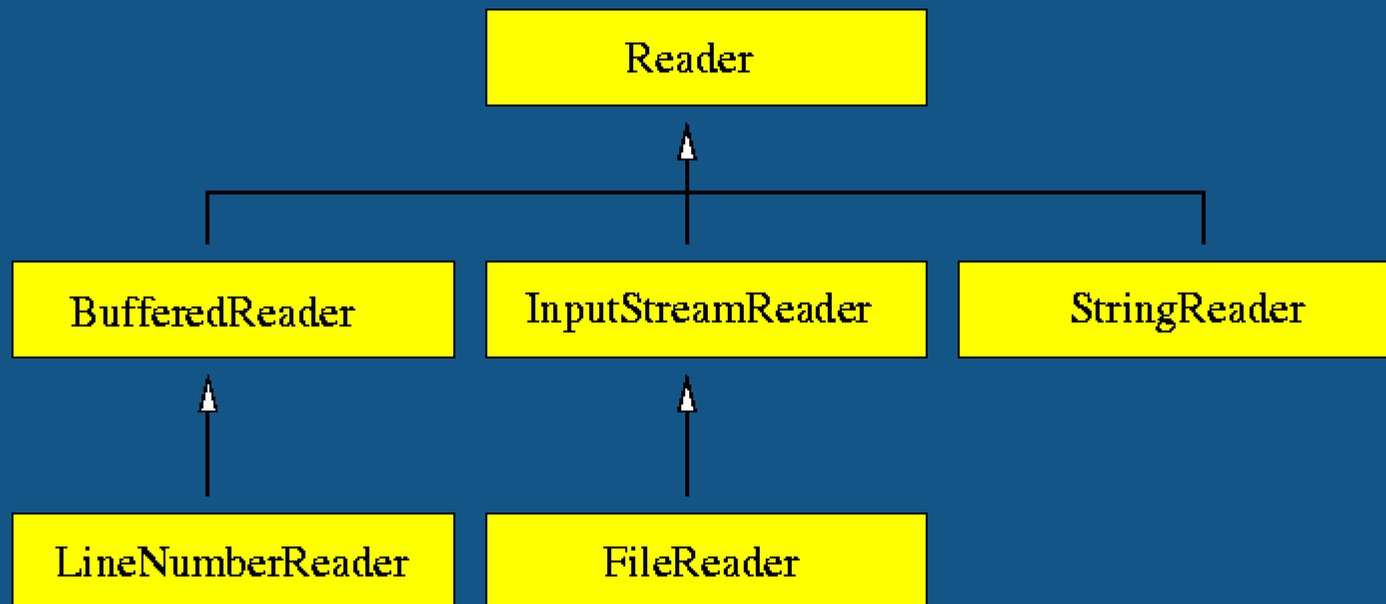
```
27
28          // Čitanje upisanih podataka. Kako nam FileInputStream
 objekt nije potreban
29// osim kao argument konstruktora DataInputStream objekta
30       ne pridružujemo mu referencu (ostaje bezimen)
31
32          DataInputStream dis = new DataInputStream(new
 FileInputStream("io_example.dat")
33                  );
34
35          b = dis.readBoolean(); System.out.println("b = "+b);
36          i = dis.readInt();     System.out.println("i = "+i);
37          c = dis.readChar();    System.out.println("c = "+c);
38          s = dis.readUTF();     System.out.println("s = "+s);
39          d = dis.readDouble();  System.out.println("d = "+d);
40          dis.close();
41      }
42  }
```

NAPOMENE

- Klase `FileInputStream` i `FileOutputStream` izbacuju `FileNotFoundException` koji mora biti procesiran. Mi smo ga umjesto toga deklarisali u `throws` deklaraciji u `main` metodi. Isto tako, sve metode za čitanje i pisanje izbacuju `IOException` pa je i on deklarisan.
- Funkcije za čitanje i pisanje pojedinih ugrađenih tipova (`readInt/writeInt`, `readChar/writeChar` itd.) zapisuju/čitaju podatke binarno, u formi nezavisnoj od procesora na kojem se program izvršava.
- Java za kodiranje znakova koristi **Unicode** šemu u kojoj se svaki znak pamti u dva bajta. Na taj je način moguće kodirati 65536 znakova, dok se trenutno koristi oko 35000. Znakovi su kodirani od 0 do 65535, a prvih 256 kodova su rezervirani za ISO 8859-1 znakove (prošireni ASCII kod). S druge strane, UTF format (**Unicode Text Format)** je kreiran za zapisivanje Unicode zanakova u varijabilnom formatu koji za jedan znak može koristiti jedan, dva ili tri bajta.
- Instanciranje `DataInputStream`-a i `DataOutputStream`-a ne kreira novu datoteku. Te su klase proširenja klasa `FilterInputStream` i `FilterOutputStream` i njihova je uloga da prošire funkcionalnost stream-ova. Njihovi konstruktori kao argument uzimaju neki već otvoreni stream i na taj način konstruišu tzv. **filtrirane stream-ove**.

Tekstualne datoteke

- Pisanje i čitanje tekstualnih datoteka i uopšte `stream`-ova postavlja problem konverzije znakova između Unicode formata, koji Java interno koristi, i formata koji koristi operativni sistem.
- Za rješenje tog problema postoji drugi skup *stream-filter klasa* koje proširuju apstraktne klase Reader i Writer. Dio hijerarhije Reader-klasa je dat na sljedećoj slici:



Nastavak...

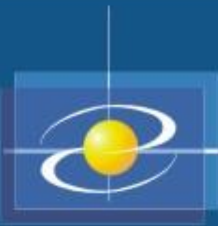
- Dio hierarhije Writer-klasa dat je na slici:

Writer

BufferedWriter OutputStreamWriter StringWriter PrintWriter

FileWriter

- Klasa `InputStreamReader` pretvara ulazni stream, koji šalje niz bajtova koji predstavljaju znakove kodirane na datoj platformi, u *reader* koji emituje Unicode znakove.
- Analogno, `OutputStreamWriter` konvertuje stream Unicode znakova u stream bajtova koji reprezentuju lokalno kodirane znakove.



Nastavak...

- Na primjer, konzola predstavlja jedan `InputStream`. Da bismo mogli čitati sa konzole treba kreirati `InputStreamReader`:

```
InputStreamReader io = new InputStreamReader(System.in);
```

- Zatim, pomoću metode `io.read()` možemo čitati *reader* znak po znak.
- Analogno, ako želimo pisati u izlazni stream (npr. datoteku) moramo ga moramo ga *filtrirati*:

```
OutputStreamWriter out = new OutputStreamWriter(new

                      FileOutputStream("output.txt") );
```

- Zatim, pomoću metode `out.write(int c)` možemo pisati u *writer* znak po znak.
- Budući da je otvaranje datoteka u tekstualnom modu vrlo česta pojava, Java nudi dvije klase koje pojednostavljuju sintaksu: to su `FileWriter` i `FileReader`. Tako je

```
FileWriter out = new FileWriter("output.txt" );
```

- Ekvivalentno sa.

```
OutputStreamWriter out = new OutputStreamWriter(new
                      FileOutputStream("output.txt") );
```

Nastavak...

- Ako ne želimo pisati u stream znak po znak stoji nam na raspolaganju klasa `PrintWriter` koja može pisati stringove i brojeve u *Writer*. Trebamo da kombinujemo `PrintWriter` sa `FileWriter`-om na ovaj način:

```
PrintWriter out = new PrintWriter(new FileWriter("output.txt"));
```

- Sada možemo koristiti metode `out.print` i `out.println` jednako kao i `System.out.print` i `System.out.println`. Pomoću tih metoda se mogu ispisati brojevi (int, short, long, float, double) znakovi, stringovi, logičke vrijednosti i objekti.
- Za čitanje nam Java ne nudi klasu poput `PrintWriter-a`. Sve što možemo napraviti je filtrirati klasu `FileReader` kroz `BufferedReader` i koristiti njenu `readLine` metodu za čitanje čitave linije teksta.

```
BufferedReader in = new BufferedReader(new FileReader("io_example.dat"));
```

- Metoda `readLine` vraća null kada više nema ulaza pa bi kod za čitanje datoteke imao ovaj oblik:

```
String line;
while((line = in.readLine()) != null)
{
        // obrada linije
}
```