Objektno orijentirano programiranje

10: Datoteke

Creative Commons

You are free to

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material

under the following terms

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** — You may not use the material for commercial purposes.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

- https://creativecommons.org/licenses/by-nc-sa/4.0/

Paketi za rad s datotekama i temeljni razred

- Potpora za rad s datotekama te općenitije ulazno/izlaznim API-jem nalazi se u dva paketa
  - Paket *java.io*
  - Paket *java.nio* (od Jave 1.4)
    - Od Jave 7 paket *java.nio* dobio je niz novih funkcionalnosti i proširenja
- Temeljna klasa: *File* (*java.io*)
  - Apstraktna reprezentacija bilo kojeg objekta datotečnog sustava (direktorija, datoteke)
  - Metode za dohvat informacija o tim objektima
  - Informacije o platformi:
    - *File.separator: String, File.separatorChar: Char, File.pathSeparator: String, File.pathSeparatorChar: Char, File[] roots = File.listRoots();*
- Novo temeljno sučelje: *Path* (*java.nio*)
  - Slično klasi *File*
  - Stari API se oslanja na *File*, noviji koriste *Path*

Kreiranje objekta tipa *File*

- *File* ima 4 konstruktora
  - *File(String pathname)*
    - npr. `new File("d:/tmp/readme.txt")`
  - *File(String parent, String child)*
    - npr. `new File("d:/tmp", "readme.txt")`
  - *File(File parent, String child)*
    - npr.
```
File dir = new File("d:/tmp");
File file = new File(dir, "readme.txt");
```
  - *File(URI uri)*
    - npr.
```
new File(new
URI("file:///d:/tmp/readme.txt"));
```
- *File* se koristi i za datoteke i za direktorije
- Stvaranje objekta tipa *File* ne mora značiti da datoteka ili direktorij postoje na disku

Primjer dohvata informacija o nekom objektu datotečnog sustava

- Prikazane su neke od metoda za dohvat informacija o nekom objektu datotečnog sustava

```
private static void showFileInfo(File file) {
                String absolutePath = file.getAbsolutePath();
                File parent = file.getParentFile();
                boolean exists = file.exists();
                boolean readable = file.canRead();
                boolean writeable = file.canWrite();
                boolean executable = file.canExecute();
                long fileSize = file.length();
                boolean isFile = file.isFile();
                boolean isDirectory = file.isDirectory();
                boolean isHidden = file.isHidden();
...
```

10_InputOutput/…/hr/fer/oop/io/FileInfoExample.java

Ostale metode razreda *File*

- *File* sadrži još nekoliko drugih metoda
  - statičke metode za stvaranje privremenih datoteka (ime datoteke nam nije bitno)
  - stvaranje datoteke samo ako takva već ne postoji
  - preimenovanje datoteke, brisanje
  - stvaranje direktorija / poddirektorija
  - apsolutne staze / kanonske staze
  - informacije o particiji
  - dohvat sadržaja nekog direktorija
  - …

Unaprjeđenje od Jave 7 - *File* i *Path*

- Apstraktna staza do elemenata datotečnog sustava predstavljena je sučeljem *Path* (paket *java.nio.file*)
  - *npr.* `Path p = Path.of("d:/tmp/readme.txt");`
- Klasa *Paths* je stari API s metodama za stvaranje objekata tipa *Path*
  - npr. `Path p = Paths.get("d:/tmp/readme.txt");`
- Razred *Files* sastoji se samo od statičkih metoda za dohvat informacija o objektima tipa *Path* i niza korisnih metoda, npr.
  - kopiranje datoteka, stvaranje direktorija, datoteka i simboličkih linkova, premještanje datoteka i brisanje …
- Moguće je preslikavanje u oba smjera
  - Primjerci razreda *File* imaju metodu *toPath()*
    - `File f = new File("..."); Path p = f.toPath();`
  - Primjerci razreda *Path* imaju metodu *toFile()*
    - `Path p = Paths.get("..."); File f = p.toFile();`
- Prethodni primjer moguće raspisati s novim API-em kao u

10_InputOutput/…/hr/fer/oop/io/PathInfoExample.java

Ispis sadržaja direktorija

- Metoda *Files.newDirectoryStream(Path)* vraća *DirectoryStream<Path>*
  - Sučelje *DirectoryStream<T>* nasljeđuje
    - *Iterable<T>* – za prolazak kroz elemente direktorija i
    - *Closeable* – tok se treba zatvoriti nakon korištenja
    - Lagano se koristi u kombinaciji s *try-with-resources* i petljom *for-each*
- Metoda Files.newDirectoryStream je preopterećena npr.
  - *Files.newDirectoryStream(Path, DirectoryStream.Filter<? super Path>)*
  - Sučelje *DirectoryStream.Filter* služi za filtriranje. Ima sljedeću metodu:
    - *boolean accept(T entry)* – ako vrati laž onda se taj put preskače

Primjer filtera temeljem zadanih ekstenzija

- Filter uključuje samo datoteke s traženim ekstenzijama

```
public class FilterByExtensions implements Filter<Path> {
  private final Set<String> extensions;
  public FilterByExtensions(String... extensions) {
       this.extensions = Set.of(extensions);
  }
  @Override
  public boolean accept(Path entry) throws IOException {
       String filename = entry.getFileName().toString();
       int ind = filename.lastIndexOf('.');
       if (ind != -1) {
              String ext = filename.substring(ind + 1);
              return extensions.contains(ext);
       }
       else return false;
   }
}
```

10_InputOutput/…/hr/fer/oop/io/FilterByExtensions.java

Primjer ispisa sadržaja direktorija

10_InputOutput/…/hr/fer/oop/io/DirContent.java

- Filter uključuje samo datoteke s traženim ekstenzijama

```
Scanner sc = new Scanner(System.in);
System.out.println("Enter directory:");
String dirName = sc.nextLine();
Path directory = Path.of(dirName);

DirectoryStream.Filter<Path> filter =
                    new FilterByExtensions("txt", "pdf", "pptx");

try(DirectoryStream<Path> dirStream =
             Files.newDirectoryStream(directory, filter)) {
   for(Path path : dirStream){
       System.out.printf("%s (%s bytes) (%s) %n",
                    path.getFileName().toString(),
                    Files.size(path),
                    Files.getLastModifiedTime(path).toString() ...
```

Filter za ispis podstabla

- Pretpostavimo da želimo potražiti sve java ili class datoteke, ali u cijelom podstablu, a ne samo u trenutnom direktoriju.
- U filter je potrebno uključiti i direktorije
  - Obratiti pažnju da je u primjeru *endsWith* iz klase *String*, a ne iz *Path*. Sučelje *Path* definira endsWith koja uspoređuje dijelove putanje, a ne tekst putanje.

```
public class MyPathStreamFilter implements Filter<Path> {

	@Override
	public boolean accept(Path entry) throws IOException {
		String stringPath = entry.toString();
		return stringPath.endsWith(".java") ||
			stringPath.endsWith(".class") ||
				Files.isDirectory(entry);
	}
}
```

10_InputOutput/…/hr/fer/oop/io/MyPathStreamFilter.java

Rekurzivni ispis podstabla

- Ispis podstabla se svede na rekurzivne pozive za svaki pronađeni direktorij
- Izvedivo (**10_InputOutput/…/hr/fer/oop/io/DirTree.java**) ali postoji bolji način korištenjem *Visitora*

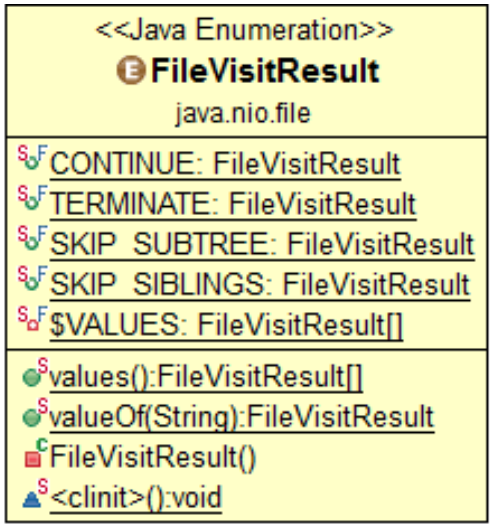
```
Enter directory:
D:\GitRepositories\FER-OOP\10_InputOutput
D:\GitRepositories\FER-OOP\10_InputOutput
|.settings
|src
|-main
|--java
|---hr
|----fer
|-----oop
|------io
|       DirTree.java (1150 bytes)
|       FileInfoExample.java (1783 bytes)
|       MyFilenameFilter.java (315 bytes)
|------iostreams
|       CustomDecoratorExample.java (1009 bytes)
```

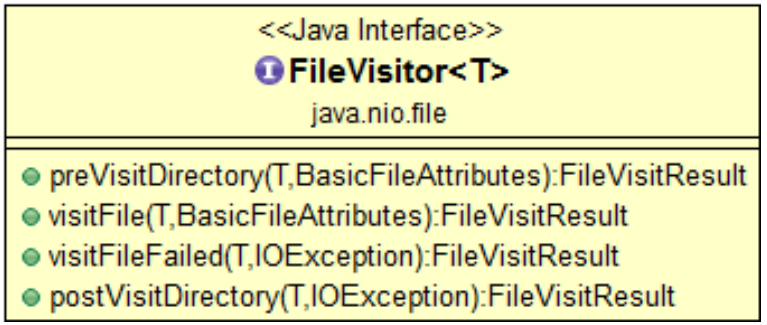
```
Path root = Path.of(dirName).toAbsolutePath();
directoryTree(root, 0);

public static void directoryTree(Path directory, int level) {
  ...
  DirectoryStream.Filter<Path> filter = new MyPathStreamFilter();
  try(DirectoryStream<Path> dirStream =
                  Files.newDirectoryStream(directory, filter)) {
      for(Path path : dirStream){
      if (Files.isDirectory(path)){
            directoryTree(path, level + 1);
      }
      else ... ispiši datoteku ...
```

Obilazak podstabla datotečnog sustava sučeljem *FileVisitor*

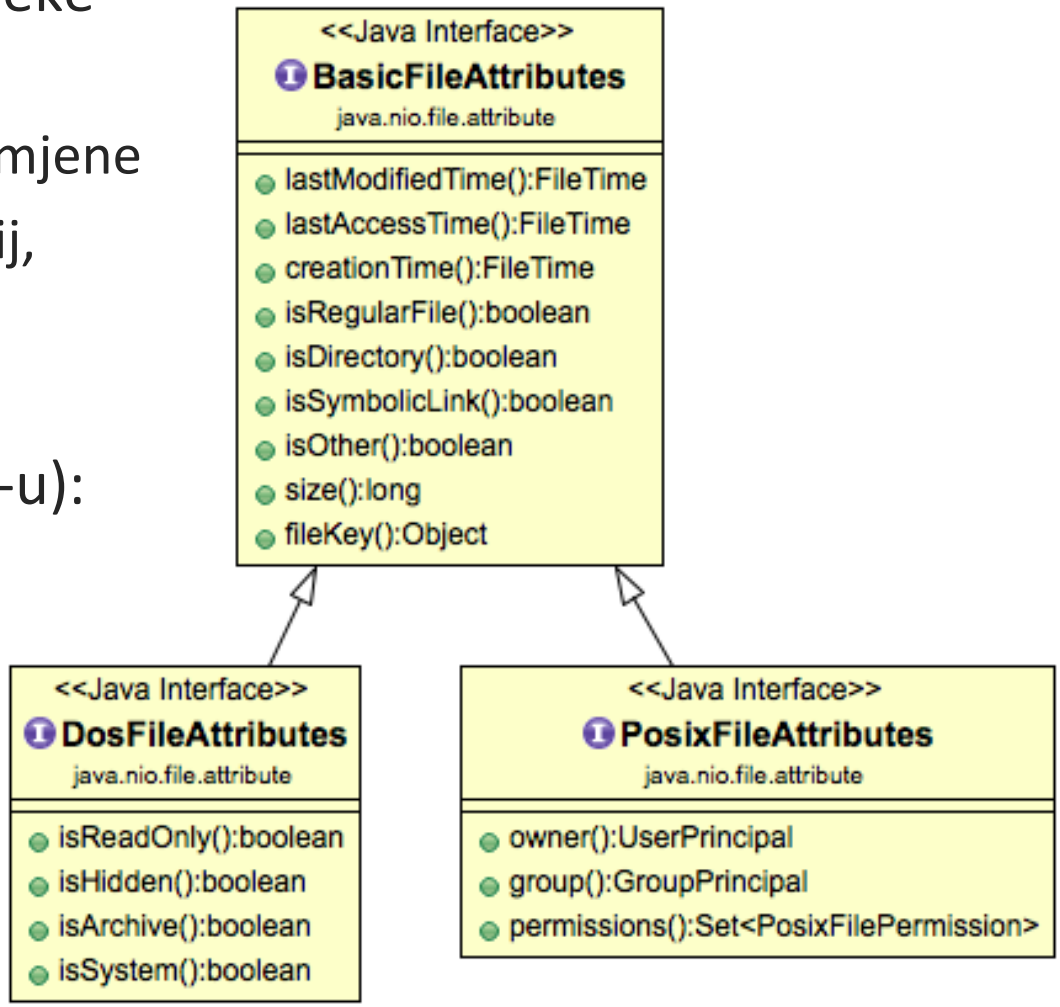
- U prethodnom primjeru smo sami radili dohvat sadržaja direktorija i rekurzivno obilazili podstablo
- Razred *Files* nudi metodu *walkFileTree*(path, visitor) koja obilazi čvorove podstabla zadane staze i za svaki čvor obavlja određeni posao
- Posao koji treba obaviti modeliran je zasebnim sučeljem *FileVisitor*
  - Implementaciju ovog obilaska ne zanima što će korisnik napraviti s posjećenim elementima (ispisati ih na ekran, u datoteku, zbrojiti veličine, …)
- Ovakav način obilaska elemenata je opisan oblikovnim obrascem *Visitor*





Klasa *java.nio.file.attribute.BasicFileAttributes*

- Predstavlja svojstva datoteke koja su joj dodijeljena:
  - Vremena stvaranja i promjene
  - Vrsta datoteka (direktorij, poveznica,…)
  - Veličina
- Dvije podvrste (ovisi o OS-u):
  - Dos (Windows)
  - Posix (Unix, Linux)



Obilazak stabla metodom *walkFileTree*

- Potrebno je stvoriti primjerak razreda koji implementira sučelje *FileVisitor<Path>* i pozvati metodu *Files.walkFileTree*
  - Metoda *walkFileTree* šeta po podstablu i zove odgovarajuće metode *FileVisitora*

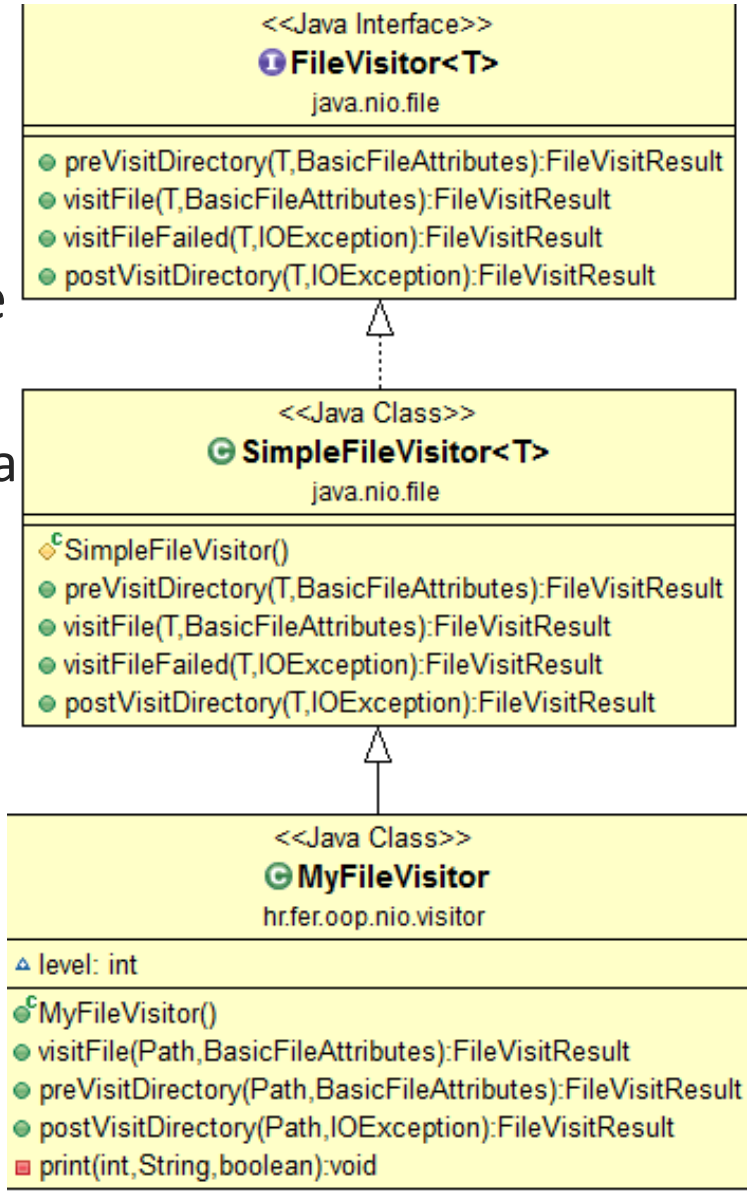
10_InputOutput/…/hr/fer/oop/visitor/Main.java

```
public static void main(String[] args) {
	...
	String dirName = ...
	FileVisitor<Path> visitor = new MyFileVisitor();
	Path path = Paths.get(dirName);
	try {
		Files.walkFileTree(path, visitor);
	} catch (IOException e) {
		e.printStackTrace();
	}
}
```

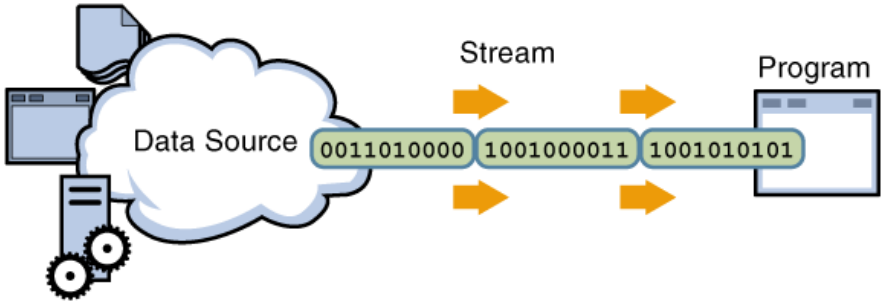
Implementacija *FileVisitora*

- Umjesto direktne implementacije sučelja *FileVisitor* može se iskoristiti postojanje jednostavne implementacije *SimpleFileVisitor*
  - metode te implementacija ne rade ništa značajno, ali nam štede trud ako ne želimo implementirati sve metode
  - Metode koje su nam bitne nadjačamo
    - U ovom primjeru *visitFile*, *preVisitDirectory* i *postVisitDirectory*
- Pogledati kôd *MyFileVisitora*

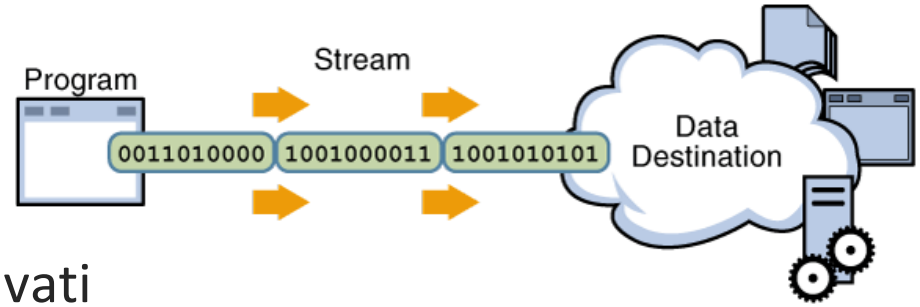
10_InputOutput/…/hr/fer/oop/visitor/MyFileVisitor.java



I/O tokovi



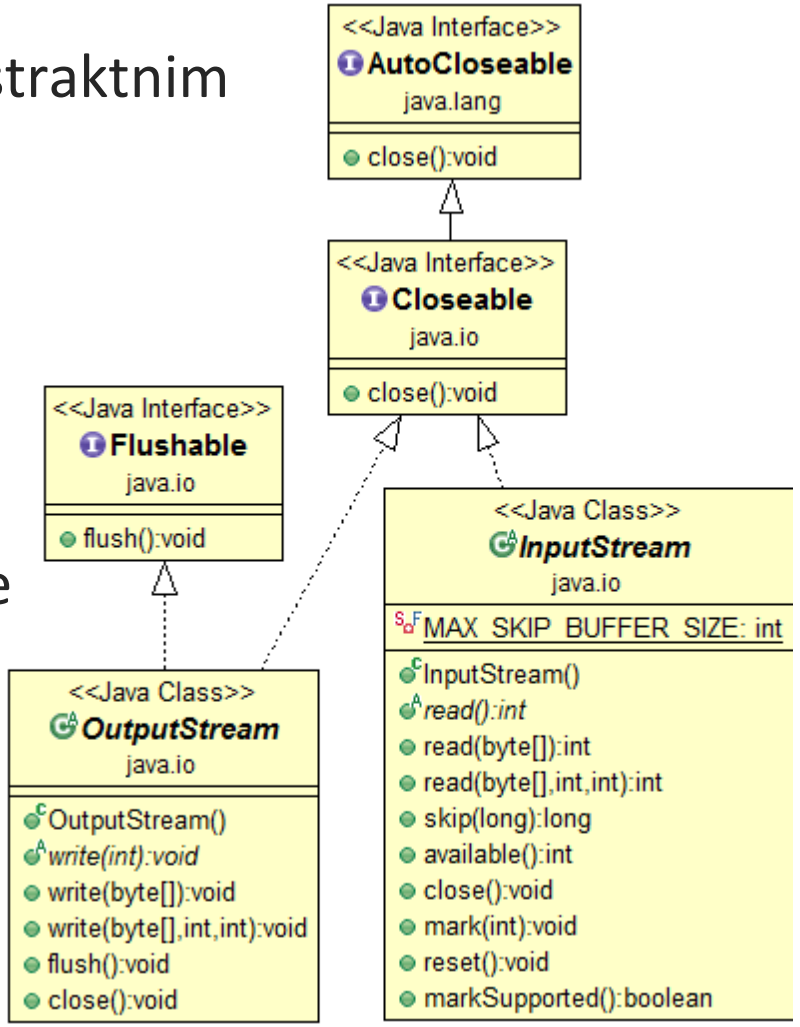
- Tokovi su jednosmjerni
- Ulazni tok
  - ponaša se kao izvor podataka
  - klijent može čitati podatak po podatak (ili više njih slijedno u spremnik)
- Izlazni tok
  - ponaša se kao ponor podataka
  - klijent u njega može samo zapisivati



- Paket java.io podržava dvije vrste tokova podataka
  - tokovi okteta: podaci s kojima radimo su okteti (*byte*) - prikladno za rad s binarnim podatcima
  - tokovi znakova: podaci s kojima radimo su znakovi (*char*) - prikladno za rad s tekstovnim podatcima

Tokovi okteta

- Izvor i ponor okteta modelirani su apstraktnim razredima:
  - *InputStream*
  - *OutputStream*
- *InputStream* nudi metode za čitanje okteta odnosno polja okteta.
- *OutputStream* nudi metode za pisanje okteta odnosno polja okteta.
  - Implementira sučelja *Closeable* i *Flushable* (sučelje za resurse koji se mogu prazniti)



Zašto apstraktno modeliranje tokova?

- Izvor i ponor mogu biti bilo što:
  - Datoteka na disku računala
  - TCP priključna točka s kojom preko mreže razgovaramo s drugom aplikacijom
  - Potprogram koji na zahtjev generira tražene podatke (npr. *InputStream* koji vraća slučajne brojeve)
- Neke konkretne implementacije:
  - *FileInputStream*, *FileOutputStream* (čitanje i pisanje u datoteku)
    - Za čitanje iz datoteke možemo direktno instancirati primjerak *FileInputStream*-a ili možemo koristiti statičku metodu *Files.newInputStream*
  - *ByteArrayInputStream*, *ByteArrayOutputStream* (čitanje i pisanje u zadani spremnik)

Primjer čitanja sadržaja binarne datoteke

10_InputOutput/…/hr/fer/oop/iostreams/DumpBinaryFile.java

```
public class DumpBinaryFile {
  public static void main(String[] args) {
        Path p = Paths.get("D:/temp/photo.jpg");
        try (InputStream is =
                  Files.newInputStream(p, StandardOpenOption.READ)) {
           byte[] buff = new byte[1024];
           while (true) {
                int r = is.read(buff);
                if (r < 1) break;
                for(int i=0; i<r; i++)
                       System.out.format("%02x ", buff[i]);
           }
        } catch (IOException ex) {
                System.err.println(ex);
        }
  }
}
```

Kombiniranje različitih ponašanja nekog toka

- Neovisno o konkretnom izvoru/ponoru okteta, može se modificirati ponašanje izvora/ponora na različite načine, npr. da podržava
  - *bufferirano* čitanje/pisanje, kriptiranje/dekriptiranje u letu, komprimiranje/dekomprimiranje u letu, …
  - tako se npr. može izgraditi ponor podataka koji će biti *bufferiran* i koji će generirati ZIP-ani sadržaj podataka koji mu se šalju
- Kako se različite mogućnosti moraju moći kombinirati proizvoljno, prikladan oblikovni obrazac za rješavanje ovakvog problema je *dekorator*
  - Prije korištenja ugrađenih dekoratora ovaj način rada s tokovima bit će ilustriran na primjeru izlaznog toka koji za svaki primljeni oktet na izlaz ispisuje originalni oktet XOR-an s brojem x koji je zadan u konstruktoru takvog toka.

Dekorator na primjeru toka podataka

- Svaki ponor okteta izveden je iz apstraktne klase *OutputStream*
  - konkretni ponor okteta je npr. klasa *FileOutputStream* koji nasljeđuje klasu *OutputStream* i oktete zapisuje u datoteku
- Definiramo novu klasu *ScrambledOutputStream* koji također nasljeđuje *OutputStream* i preko konstruktora prima referencu na postojeći *OutputStream* kojem će prosljeđivati izmijenjene oktete
  - *OutputStream* je apstraktna klasa pa je potrebno implementirati metodu *write*, a ostale se nadjačavaju prema potrebi.
    - Metoda *write* prima parametar tipa int kojem zadnjih 8 bitova predstavlja oktet koji treba zapisati u izlazni tok.

Razred *ScrambledOutputStream* kao primjer dekoratora toka

```
public class ScrambledOutputStream extends OutputStream {
  private OutputStream stream;
  private byte x;

  public ScrambledOutputStream(OutputStream stream, byte x) {
    this.stream = stream;
    this.x = x;
  }
  @Override
  public void write(int b) throws IOException {
    stream.write(b ^ x);
  }

  @Override
  public void close() throws IOException {
    stream.close();
  }
}
```

10_InputOutput/…/hr/fer/oop/iostreams/ScrambledOutputStream.java

Primjer korištenja dekoriranog toka

- Koristimo try-with-resources (*OutputStream* je *Closeable*)

```
private static void writeFile(String filename) {
  try (OutputStream os = new ScrambledOutputStream(
         new FileOutputStream(filename), (byte) 0xC3)) // 1100 0011
  {
    os.write(150); //0x96 (1001 0110)
    os.write(new byte[] { 35, 70, 120 }); //0x23 (0010 0011)
     // 0x46 (0100 0110) 0x78 (0111 1000)
    os.write(129); //0x81 (1000 0001)
  }
  catch (IOException e) {
    e.printStackTrace();
  }
}
```

10_InputOutput/…/hr/fer/oop/iostreams/CustomDecoratorExample.java

- U datoteku će biti zapisano (binarno)

```
1010101 11100000 10000101 10111011 1000010
```

Gotovi dekoratori za tokove

- U *java.io* već imamo niz dekoratora:
  - *BufferedInputStream*
  - *DataInputStream*
  - *ObjectInputStream*
  - *PushbackInputStream*
  - *SequenceInputStream*
- U *java.util.zip* se još nalazi *ZipInputStream*

10_InputOutput/…/hr/fer/oop/iostreams/ZipExample.java

- Slično je i s dekoratorima razreda *OutputStream*
- Znakovni tokovi su dekoratori tokova okteta
- Dekoratori se mogu kombinirati po želji, npr.

```
OutputStream os = new ZipOutputStream(new BufferedOutputStream(
  new ScrambledOutputStream(new FileOutputStream("file.dat"),
  (byte) 0xC3)));
```

Znakovni tokovi

- U Javi: okteti ≠ znakovi (za razliku od C-a)
- Da bismo znali kako znakove kodirati u oktete, trebamo znati koju ćemo kodnu stranicu koristiti
  - kodna stranica je u Javi modelirana razredom *Charset* (paket *java.nio.charset*)
  - Na svim Javinim platformama automatski su podržane i dostupne sljedeće kodne stranice:
    - US-ASCII, ISO-8859-1
    - UTF-8, UTF-16BE, UTF-16LE, UTF-16
  - Razred *StandardCharset* omogućava dohvat svih tih kodnih stranica
    - `Charset c = StandardCharsets.UTF_8;`

Znakovni tokovi

- Alternativno, ako znamo ime kodne stranice, možemo koristiti i poziv:

```
Charset c2 = Charset.forName("ISO-8859-1");
```

- Tako možemo doći i do nestandardno podržanih kodnih stranica (ako su instalirane); inače iznimka
- Jednom kad imamo kodnu stranicu, konverzija okteta u znakove ide ovako:

```
Charset c = StandardCharsets.UTF_8;
Charset c2 = Charset.forName("ISO-8859-2");
byte[] bytes = new byte[] {-59, -95, -60, -111, -60,
     -115, -60, -121, -59, -66};
String text = new String(bytes, c); // šđčćž po UTF8
byte[] bytes2 = text.getBytes(c2); //-71 -16 -24 -26 -66
                                   // & 0xFF  185 240 232 230 190
```

10_InputOutput/…/hr/fer/oop/iostreams/EncodingExample.java

Znakovni tokovi

- Znakovni tokovi unutar paketa java.io modelirani su apstraktnim razredima *Reader* i *Writer*
- Metode ovih razreda su slične metodama *InputStream* i *OutputStream* samo što umjesto okteta i polja okteta primaju znakove i polja znakova
- Postoji nekoliko konkretnih implementacija
  - *FileReader* i *FileWriter* (koriste pretpostavljenu kodnu stranicu!)
  - *StringReader* i *StringWriter*
  - *CharArrayReader* i *CharArrayWriter*
- Postoji nekoliko dekoratora: *BufferedReader, BufferedWriter, LineNumberReader, PushbackReader*

Veza znakovnih tokova i tokova okteta

- Konačno, postoji most između znakovnih tokova i tokova okteta
- *InputStreamReader*
  - *reader* koji oktete čita iz toka okteta na koji je spojen, oktete dekodira uporabom zadane kodne stranice i time generira znakove
- *OutputStreamWriter*
  - *writer* koji iz znakova generira oktete temeljem zadane kodne stranice

Uobičajeni idiomi za rad s tekstovnim datotekama

- Često korišteni idiom za rad s tekstovnim datotekama

```
BufferedReader br = new BufferedReader(
  new InputStreamReader(
    new BufferedInputStream(
      new FileInputStream("name.txt")),"UTF-8"));
String line = br.readLine();

Writer bw = new BufferedWriter(
  new OutputStreamWriter(
    new BufferedOutputStream(
      new FileOutputStream("name2.txt")),"UTF-8"));
bw.write(line);
```

Pomoćne (korisne) metode

- Pomoćne (korisne) metode

```
Charset c = StandardCharsets.UTF_8;
Path path = Path.of("name.txt");
List<String> lines = Files.readAllLines(path, c);
byte[] content = Files.readAllBytes(path);
InputStream is = Files.newInputStream(path);
OutputStream os = Files.newOutputStream(
                        path,
                        StandardOpenOption.CREATE_NEW
                        // CREATE,APPEND,WRITE,TRUNCATE_EXISTING
                        // see Javadoc of this enum for more info
                    );
```

Datoteke sa slučajnim pristupom

- Iako često korištena, apstrakcija tokova nije primjenjiva na sve zadatke za koje koristimo datoteke
- Za dobivanje datoteke sa slučajnim pristupom postoji razred *RandomAccessFile*, koji nudi metode tipa:
  - *getFilePointer()* i *seek(position)*
- Veza prema “C”-olikom API-ju za datoteke