**I/O introducere**

Java contine o bogata multime de biblioteci ce efectueaza functii de intrare/iesire. Java defineste un canal de intrare-iesire ca pe un flux (stream). Fluxul este o sursa de intrare sau o destinatie de iesire. Sursele si destinatiile pot fi de multiple feluri de date precum octeti, tipuri primitive, caractere si obiecte. Unele fluxuri doar trimit date altele le manipuleaza si le transforma intr-un mod usor de interpretat.

Un program utilizeaza un flux de intrare pentru a citi date de la o sursa, un item la un moment dat. Acelasi lucru se intampla cu fluxurile de iesire.

Indiferent cum lucreaza ele intern, toate fluxurile reprezinta o secventa de date.

Sursele si destinatiile se numesc fluxuri nod. Spre exemplu: fisiere, memorie, pipes intre fire sau procese.

Un dezvoltator de aplicatie utilizeaza in mod uzual fluxurile I/O pentru a citi sau scrie fisiere, pentru a citi sa scrie informatii de la sau la dispozitive de iesire (tastatura – intrarea standard, consola – iesirea standard). O aplicatie foloseste un socket pentru a comunica cu o alta aplicatie sau sistem la distanta.

Java suporta doua tipuri de stream-uri: caracter si byte. Intrarea si iesirea datelor caracter sunt manipulate de readers si writers, iar a datelor octet de input streams si output streams.

Mai precis, intrarile pentru fluxuri de octeti sunt gestionate de subclase ale lui InputStream iar iesirile de subclase ale OutputStream. La nivelul caracterelor (codificate Unicode) avem clasele de baza Reader si Writer.

Fluxurile de octeti sunt in general folosite pentru a citi fisiere imagine, fisiere audio si obiecte.

Cele trei metode de baza pentru citire din InputStream sunt:

* int read();, returneaza fie un byte citit din flux fie -1, ce indica atingerea sfarsitului de fisier
* int read(byte[] b);, citeste fluxul intr-un sir de octeti si returneaza numarul de octeti cititi
* int read(byte[] b, int off, int len);, la fel ca anterioara doar ca ultimele doua argumente indica un subdomeniu al sirului, care urmeaza a fi umplut

Pentru eficienta se recomanda sa citim date in blocuri cat mai mari sau sa folosim buffere de flux.

Dupa folosire un flux trebuie inchis. In Java SE 7 InputStream implementeaza AutoCloseable.

Alte metode:

* int available();, returneaza numarul de octeti ce sunt disponibili imediat pentru a fi citite din flux. O operatie de citire ce urmeaza acestui apel poate returna mai multi octeti
* long skip(long n);, ignora numarul specificat de octeti din flux
* boolean markSupported(); void mark(int readLimit); void reset();, efectueaza operatii push-back pe un stream, daca strimul suporta aceste operatii. markSupported() returneaza true daca metodele mark() si reset() sunt operationale pe fluxul curent. mark() indica faptul ca punctul curent din flux va fi marcat si un buffer suficient de mare cat cel putin numarul de octeti specificat de argument va fi creat. Parametrul metodei specifica numarul de octeti ce pot fi recititi apeland reset().

Cele trei metode de scriere din OutputStream sunt:

* void write(int c);
* void write(byte[] buffer);
* void write(byte[] buffer, int offset, int length);

alte metode:

* void close();
* void flush();, care forteaza scrierea in flux

Ca si la citire este recomandata scrierea datelor in blocuri cat mai mari.

In mod asemanator avem metodele de citire din Reader:

* int read();, returneaza fie un int citit din flux, reprezentand un caracter Unicode, fie -1, ce indica atingerea sfarsitului de fisier
* int read(char[] b);, citeste fluxul intr-un sir de caractere si returneaza numarul de octeti cititi
* int read(char[] b, int off, int len);, la fel ca anterioara doar ca ultimele doua argumente indica un subdomeniu al sirului, care urmeaza a fi umplut

Iar pentru scriere metodele din Writer sunt:

* void write(int c);
* void write(char[] buffer);
* void write(char[] buffer, int offset, int length);
* void write(String string);
* void write(String string, int offset, int length);

Fie urmatorul exemplu de copiere a unui flux de caractere, dat intr-un fisier in.txt.

**public** **class** CharStreamCopyTest {

**public** **static** **void** main(String[] args) {

**char**[] c = **new** **char**[128];

**int** cLen = c.length;

**try** (FileReader fr = **new** FileReader("in.txt");

FileWriter fw = **new** FileWriter("out.txt")) {

**int** count = 0;

**int** read = 0;

**while** ((read = fr.read(c)) != -1) {

**if** (read < cLen)

fw.write(c, 0, read);

**else**

fw.write(c);

count += read;

}

System.*out*.println("Wrote: " + count + " characters.");

} **catch** (FileNotFoundException f) {

System.*out*.println("File not found: " + f);

} **catch** (IOException e) {

System.*out*.println("IOException: " + e);

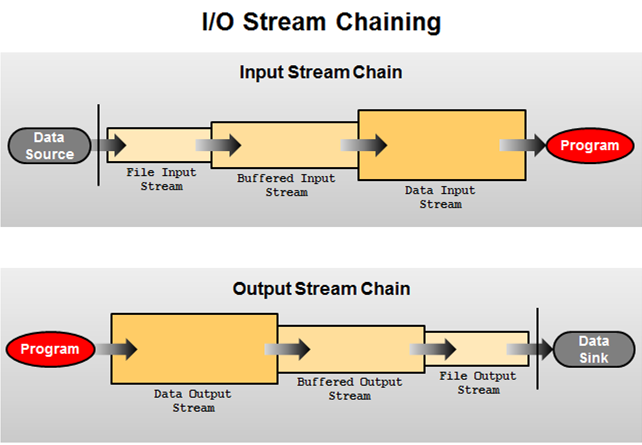
}

}

}

Copierea se face folosind un sir de caractere. FileReader si FileWriter sunt clase destinate citirii si scrierii fluxurilor de caractere, precum fisierele text.

**Inlantuirea fluxurilor I/O**



Un program foloseste arareori un sigur obiect pentru flux, ci inlantuie o serie de stream-uri pentru a procesa date. In prima parte a figurii anterioare este dat un exemplu de flux de intrare: fluxul de fisier este operat de un buffer pentru cresterea eficientei si apoi este convertit in date. Partea a doua exemplifica drumul invers.

**public** **class** BufferedStreamCopyTest {

**public** **static** **void** main(String[] args) {

**try** (BufferedReader bufInput = **new** BufferedReader(**new** FileReader(

"in.txt"));

BufferedWriter bufOutput = **new** BufferedWriter(**new** FileWriter("out.txt"))) {

String line = "";

// read the first line

**while** ((line = bufInput.readLine()) != **null**) {

// write the line out to the output file

bufOutput.write(line);

bufOutput.newLine();

}

} **catch** (FileNotFoundException f) {

System.*out*.println("File not found: " + f);

} **catch** (IOException e) {

System.*out*.println("Exception: " + e);

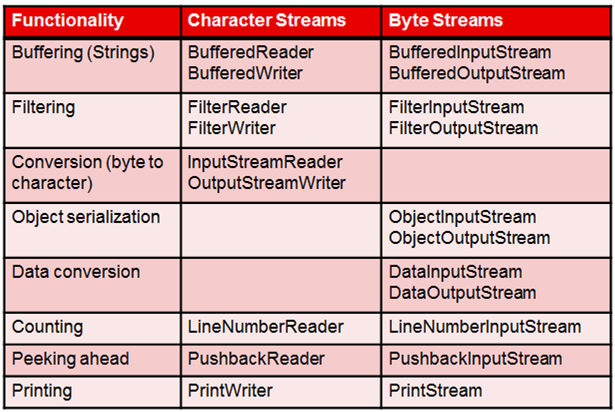
}

}

}

Fata de exemplul anterior, in loc sa citim un sir de caractere, citim o linie intreaga folosind readLine(), pe care o depunem intr-un String. Aceasta furnizeaza o foarte buna eficienta. Motivul este ca orice cerere de citire facuta de un Reader cauzeaza o cerere de citire corespunzatoare a fi facuta in fluxul de caractere. BufferReader-ul citeste caracterele din flux intr-un buffer (dimensiunea bufferului poate fi setata, dar valoarea predefinita este in general suficienta).

Un flux de procesare efectueaza o conversie catre un alt flux. Dezvoltatorul este cel care alege tipul de flux bazat pe functionalitatea pe care o avem nevoie pentru fluxul final.



**Fluxuri standard**

In clasa System din java.lang avem trei campuri instanta statice:

* out, care reprezinta fluxul de iesire standard. Acesta este in permanenta deschis si pregatit sa accepte date de iesire. In mod obisnuit acest stream corespunde monitorului sau unei alte destinatii de iesire specificata de mediul gazda sau de utilizator. Este o instanta a lui PrintStream
* in, care reprezinta fluxul de intrare standard. Acesta este in permanenta deschis si pregatit sa furnizeze date de intrare. In mod obisnuit acest stream corespunde tastaturii sau unei alte surse de intrare specificata de mediul gazda sau de utilizator. Este o instanta a lui InputStream
* err, care reprezinta fluxul de iesire standard al erorilor. Acesta este in permanenta deschis si pregatit sa accepte date de iesire. In mod obisnuit acest stream este utilizat pentru afisarea mesajelor de eroare sau a altor informatii ce ar putea veni in atentia imediata a unui utilizator. Este o instanta a lui PrintStream

In afara de obiecte PrintStream, System poate accesa instante ale java.io.Console. Obiectul Console reprezinta o consola bazata pe caracter si asociata JVM curente. Daca o masina virtuala are o consola aceasta depinde de platforma si de modul in care masina virtuala este invocata.

**public** **class** SystemConsoleTest {

**public** **static** **void** main(String[] args) {

String username = "oracle";

String password = "tiger";

**boolean** userValid = **false**;

Console cons;

// Get a Console instance

cons = System.*console*();

**if** (cons != **null**) {

String userTyped;

String pwdTyped;

**do** {

userTyped = cons.readLine("%s", "User name: ");

pwdTyped = **new** String(cons.readPassword("%s", "Password: "));

**if** (userTyped.equals(username) && pwdTyped.equals(password)) {

userValid = **true**;

} **else** {

System.*out*

.println("User name and password do not match existing credentials.\nTry again.\n");

}

} **while** (!userValid);

System.*out*.println("Success! you are now logged in.");

} **else** {

System.*out*

.println("The console is not attached to this VM. Try running this application at the command-line.");

}

}

}

Eclipse nu are atasat o consola. Exemplul anterior va trebui rulat in linia de comanda. Exemplul ilustreaza metodele clasei consola.

Scrierea la fluxul standard de iesire se face prin metodele clasei PrintStream:

* print(), care printeaza argumentul fara caracterul linie nou
* println(), care printeaza argumentul cu caracterul linie noua

Cele doua metode sunt supraincarcate pentru majoritatea tipurilor primitive (boolean, char, int, long, float si double) si pentru char[], Object si String. Daca argumentul este Object metodele apeleaza metoda toString() a argumentului. Alaturi de acestea putem folosi si metoda pentru scrierea formatata printf().

Un exemplu de citire de la tastatura prin fluxul standard de intrare este dat mai jos:

**public** **class** KeyboardInput {

**public** **static** **void** main(String[] args) {

// Wrap the System.in InputStream with a BufferedReader to read

// each line from the keyboard.

**try** (BufferedReader in = **new** BufferedReader(**new** InputStreamReader(System.*in*))) {

String s = "";

// Read each input line and echo it to the screen.

**while** (s != **null**) {

System.*out*.print("Type xyz to exit: ");

s = in.readLine().trim();

System.*out*.println("Read: " + s);

**if** (s.equals("xyz")) {

System.*exit*(0);

}

}

} **catch** (IOException e) { // Catch any IO exceptions.

System.*out*.println("Exception: " + e);

}

}

}

In instructiunea try cu resurse am deschis un BufferedReader ce este inlantuit cu un InputStreamReader, ce este inlantuit cu System.in.

Observatie: stringul null este returnat daca s-a ajuns la sfarsitul fluxului (spre exemplu utilizatorul a apasat Ctrl+C).

Un canal citeste octeti si caractere in blocuri, in loc de a citi un octet sau un caracter.

**public** **class** ByteChannelCopyTest {

**public** **static** **void** main(String[] args) {

**try** (FileChannel fcIn = **new** FileInputStream("in.txt").getChannel();

FileChannel fcOut = **new** FileOutputStream("ou.txt").getChannel()) {

System.*out*.println("File size: " + fcIn.size());

// Create a buffer to read into

ByteBuffer buff = ByteBuffer.*allocate*((**int**) fcIn.size());

System.*out*.println("Bytes remaining: " + buff.remaining());

System.*out*.println ("Bytes read: " + fcIn.read(buff));

buff.position(0);

System.*out*.println ("Buffer: " + buff);

System.*out*.println("Bytes remaining: " + buff.remaining());

System.*out*.println("Wrote: " + fcOut.write(buff) + " bytes");

} **catch** (FileNotFoundException f) {

System.*out*.println("File not found: " + f);

} **catch** (IOException e) {

System.*out*.println("IOException: " + e);

}

}

}

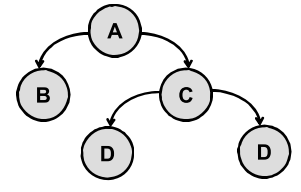
In exemplu am creat un buffer de dimensiune potrivita, buffer folosit atat la citire cat si la scriere. Citirea respectiv scrierea se fac intr-o singura operatie cu consecinte importante de performanta.

**Persistenta si serializare**

Salvarea datelor intr-un depozit permanent se numeste persistenta.

Un obiect nepersistent exista doar cat masina virtuala Java ruleaza. Serializarea este un mecanism pentru salvarea obiectelor ca un sir de octeti, care apoi pot fi reconstruiti intr-o copie a obiectului. Pentru a serializa un obiect al unei clase specifice, aceasta trebuie sa implementeze interfata java.io.Serializable. Aceasta interfata nu are metode, este o interfata marker ce indica faptul ca clasa poate fi serializata.

Cand un obiect este serializat, doar campurile obiectului sunt pastrate. Atunci cand un camp refera un obiect, campurile obiectului referit sunt de asemenea serializate (obiectul referit trebuie si el serializat). Arborele campurilor unui obiect constituie un graf obiect.



Serializarea traverseaza graful obiect si scrie datele in fluxul de iesire pentru fiecare nod al grafului.

Unele clase nu sunt serializate pentru ca ele reprezinta informatii specifice tranzitorii.

Daca un obiect graf contine o referinta neserializata, se va arunca o NotSerializableException si operatia de serializare esueaza. Campurile ce nu trebuie serializate vor fi marcate prin cuvantul rezervat transient.

Modificatorul de acces al campurilor nu influenteaza serializarea campurilor. De asemenea, campurile statice nu sunt serializate.

La deserializarea obiectelor valorile campurilor statice sunt setate la valorile declarate in clasa. Valorile campurilor nestatice tranzitorii sunt setate la valorile predefinite ale tipului.

Pe timpul serializarii un numar de versiune, serialVersionUID, este utilizat pentru a asocia iesirea serializata cu clasa utilizata in procesul de serializare. La deserializare serialVersionUID este folosit pentru verificarea faptului ca incarcarea claselor este compotibila cu obiectul ce este deserializat. Daca se observa o alta valoare a serialVersionUID, deserializarea va genera un InvalidClassException. O clasa serializata poate declara un camp static si final de tip long pentru serialVersionUID.

Daca o clasa serializata nu declara in mod explicit un serialVersionUID, atunci la rulare se va calcula o valoare predefinita pe baza diverselor aspecte ale clasei. Este, insa, puternic recomandat ca clasele serializate sa declare explicit serialVersionUID. Altfel, la deserializare pot apare InvalidCastException, pentru ca valoarea lui serialVersionUID este foarte sensibila la diversele implementari ale compilatorului Java. Este recomandat, de asemenea, ca serialVersionUID sa fie private, pentru ca nu are niciun rol in procesul de mostenire.

Fie un exemplu de folosire a unui obiect graf:

**public** **class** Stock **implements** Serializable {

**private** **static** **final** **long** *serialVersionUID* = 100L;

**private** String symbol;

**private** **int** shares;

**private** **double** purchasePrice;

**private** **transient** **double** currPrice;

**public** Stock(String symbol, **int** shares, **double** purchasePrice) {

**this**.symbol = symbol;

**this**.shares = shares;

**this**.purchasePrice = purchasePrice;

setStockPrice();

}

// This method is called post-serialization

**private** **void** readObject(ObjectInputStream ois) **throws** IOException,

ClassNotFoundException {

ois.defaultReadObject();

// perform other initiliazation

setStockPrice();

}

**public** String getSymbol() {

**return** symbol;

}

**public** **double** getValue() {

**return** shares \* currPrice;

}

// Normally the current stock price would be fetched via a feed

// Here we will simulate that

**private** **void** setStockPrice() {

Random r = **new** Random();

**double** rVal = r.nextDouble();

**double** p = 0;

**if** (currPrice == 0) {

p = purchasePrice;

} **else** {

p = currPrice;

}

// calculate the new price

**if** (rVal < 0.5) {

currPrice = p + (-10 \* rVal);

} **else** {

currPrice = p + (10 \* rVal);

}

}

@Override

**public** String toString() {

**double** value = getValue();

**return** "Stock: " + symbol + "\n" + "Shares: " + shares + " @ "

+ NumberFormat.*getCurrencyInstance*().format(purchasePrice)

+ "\n" + "Curr $: "

+ NumberFormat.*getCurrencyInstance*().format(currPrice) + "\n"

+ "Value: " + NumberFormat.*getCurrencyInstance*().format(value)

+ "\n";

}

}

**public** **class** Portfolio **implements** Serializable {

**private** **static** **final** **long** *serialVersionUID* = 101L;

**private** Set<Stock> stocks = **new** HashSet<>();

**public** Portfolio() {

}

**public** Portfolio(Stock... stocks) **throws** PortfolioException {

**for** (Stock s : stocks) {

addStock(s);

}

}

**private** **void** addStock(Stock newStock) **throws** PortfolioException {

**try** {

**if** (!stocks.add(newStock)) {

**throw** **new** PortfolioException("Stock " + newStock.getSymbol()

+ " is a duplicate.");

}

} **catch** (Exception e) {

**throw** **new** PortfolioException("Exception from Set.add method: " + e);

}

}

**public** **double** getValue() {

**double** value = 0;

**for** (Stock s : stocks) {

value += s.getValue();

}

**return** value;

}

**public** String toString() {

StringBuilder sb = **new** StringBuilder("Portfolio Summary\n");

**for** (Stock s : stocks) {

sb.append(s);

}

**return** sb.toString();

}

}

**public** **class** PortfolioException **extends** Exception {

**private** **static** **final** **long** *serialVersionUID* = 102L;

**public** PortfolioException(String message) {

**super**(message);

}

**public** PortfolioException(String message, Throwable t) {

**super**(message, t);

}

}

**public** **class** SerializeStock {

**public** **static** **void** main(String[] args) {

// Create a stock portfolio

Stock s1 = **new** Stock("ORCL", 100, 32.50);

Stock s2 = **new** Stock("APPL", 100, 245);

Stock s3 = **new** Stock("GOGL", 100, 54.67);

Portfolio p = **null**;

**try** {

p = **new** Portfolio(s1, s2, s3);

} **catch** (PortfolioException pe) {

System.*out*.println("Exception creating Portfolio: " + pe);

System.*exit*(-1);

}

System.*out*.println("Before serializaton:\n" + p + "\n");

// Write out the Portfolio

**try** (FileOutputStream fos = **new** FileOutputStream("test.out");

ObjectOutputStream out = **new** ObjectOutputStream(fos)) {

out.writeObject(p);

System.*out*.println("Successfully wrote Portfolio as an object");

} **catch** (IOException i) {

System.*out*.println("Exception writing out Portfolio: " + i);

}

// Read the Portfolio back in

**try** (FileInputStream fis = **new** FileInputStream("test.out");

ObjectInputStream in = **new** ObjectInputStream(fis)) {

Portfolio newP = (Portfolio) in.readObject();

System.*out*.println("Success: read Portfolio back in:\n" + newP);

} **catch** (ClassNotFoundException | IOException i) {

System.*out*.println("Exception reading in Portfolio: " + i);

}

}

}

Clasa Portfolio este formata dintr-o multime de Stock-uri. Pe timpul serializarii currPrice nu este serializat, fiind marcat cu transient. Am creat o metoda care sa seteze valoarea campului tranzitoriu.

In clasa de test am creat un FileOuputStream inlantuit cu un ObjectOutputStream. Aceasta permite ca octetii generati de ObjectOutputStream sa fie scrisi intr-un fisier folosind metoda writeObject(). Aceasta metoda parcurge graful si scrie datele continute in campurile non-transient si non-static.

Invers, am creat un FileInputStream inlantuit cu un ObjectInputStream. Octetii sunt cititi de readObject() si obiectul este refacut pentru campurile non-statice si non-transient. Metoda returneaza un Object ce trebuie convertit la tipul cerut.

Un obiect ce este serializat si deserializat isi poate controla serializarea campurilor. Metoda writeObject() este invocata pe obiectul ce urmeaza a fi serializat. Daca obiectul nu contine o astfel de metoda, metoda defaultWriteObject() este invocata. Metoda defaultWriteObject() trebuie apelata o data si doar o data de catre metoda writeObject() a obiectului serializat.

Pe timpul deserializarii metoda readObject() este invocata pe obiectul ce se deserializeaza. Semnatura metodei este importanta (vezi exemplul din clasa Stock). In aceasta clasa am furnizat metoda readObject() pentru a ne asigura ca currPrice este setat dupa deserializarea obiectului.

Observatie: currPrice este setat si in constructor, dar constructorul nu este apelat pe timpul deserializarii.

**Clasa java.io.File**

Pachetul java.io permite programelor Java sa citeasca si sa scrie date in diverse formate (texte, grafice, sunete si video). Pachetul contine clase ce permit accesul la date secvential sau aleator.

Clasele Reader si Writer precum si subclasele lor sunt utilizate pentru acces secvential, adica un flux secvential de biti.

Clasa RandomAccessFile si subclasele sale sunt folosite pentru a scrie sau citi aleator datele din fisier.

In exemplul urmator am creat si apoi am adaugat, intr-un fisier cu acces aleator, mai multe intrari editate intr-un TextField:

**import** java.io.\*;

**import** java.awt.\*;

**import** java.awt.event.\*;

**import** javax.swing.\*;

**public** **class** SampleWriteFile **implements** ActionListener {

**private** Frame f;

**private** Button write, clear, exit;

**private** TextField txt;

**public** SampleWriteFile(String s) {

f=**new** Frame(s);

write=**new** Button("Scrie in fisier");

write.addActionListener(**this**);

clear=**new** Button("sterge");

clear.addActionListener(**this**);

exit=**new** Button("iesire");

exit.addActionListener(**this**);

txt=**new** TextField(20);

Label l=**new** Label("introdu un text");

Panel p=**new** Panel();

Panel p2=**new** Panel();

p.add(l);p.add(txt);

p2.add(write);p2.add(clear);p2.add(exit);

f.add(p, "North");

f.add(p2);

f.pack();

f.setResizable(**false**);

f.setVisible(**true**);

}

**public** **static** **void** main(String args[]) {

**new** SampleWriteFile("creare fisier");

}

**public** **void** actionPerformed(ActionEvent e) {

**if**(e.getSource()==clear) {

txt.setText("");

}

**if**(e.getSource()==exit) {

System.*exit*(0);

}

**if**(e.getSource()==write) {

writeFile();

}

}

**public** **void** writeFile() {

**try** {

RandomAccessFile file=**new** RandomAccessFile("sampletext.txt", "rw");

file.seek(file.length());

file.writeBytes(txt.getText()+ "\n");

}

**catch**(IOException e) {

JOptionPane.*showMessageDialog*(**null**,"nu putem scrie in fisier","Exception",JOptionPane.*ERROR\_MESSAGE*);

}

}

}

Metoda seek() a fost utilizata pentru a specifica locatia din fisier in care are loc o anumita operatie. In exemplul nostru am facut pozitionarea la sfarsitul fisierului. Metoda writeBytes() este folosita pentru a scrie datele in fisier.

Clasa java.io.File poate reprezenta un director sau un fisier. Aceasta permite navigarea, descrierea si accesul la fisiere si directoare. Managerul de securitate Java permite doar acele operatii care sunt in interiorul unui context de securitate. Deoarece majoritatea browserelor nu permit orice tip de acces, clasa File si clasele inrudite sunt utilizate in aplicatii si nu in applet-uri.

Crearea unui obiect File nu inseamna automat crearea unui fiser sau director real. Si stergerea unui obiect File de catre garbage collector nu inseamna neaparat stergerea fizica a acestuia. Constructorii clasei sunt:

* File (File directoryObj, String fileName), creaza o noua instanta utilizand path-ul unui fisier (director) existent, dat ca prim parametru, si avand numele dat de al doilea parametru
* File (String pathName), creaza o instanta avand numele si calea date de parametru
* File (String pathName, String fileName) , creaza o noua instanta utilizand path-ul dat de primul parametru, si avand numele dat de al doilea parametru

Avem urmatoarele metode in File:

* exists(), care confirma daca fisierul specificat exista
* getName(), returneaza numele fisierului sau directorului (ultimul element al unui cai complete)
* getAbsolutePath(), returneaza un string ce este calea absoluta a unui fisier sau director
* isDirectory()
* isFile()
* canRead()
* getTotalSpace(), returneaza un long ce reprezinta dimensiunea in bytes a partitiei numita de calea abstracta
* getFreeSpace(), returneaza un long ce reprezinta numarul de bytes disponibili ai partitiei numita de calea abstracta. Valoarea nu este neaparat certa.
* setWritable(), permite unui fisier sa poata fi scris. Primul parametru boolean, daca are valoarea true, permite scrierea. Al doilea parametru boolean da permisiune de scriere doar celui care a creat sau este proprietarul fisierului, daca valoarea este true, altfel tuturor
* setExecutable(), permite unui fisier sa se execute operatii pe el. Primul parametru boolean, daca are valoarea true, permite acest lucru. Al doilea parametru boolean da permisiunea doar celui care a creat sau este proprietarul fisierului, daca valoarea este true, altfel tuturor
* canExecute(), returneaza true daca fisierul poate fi executat

In exemplul urmator am definit o functie de cautare a unui fisier intr-un director specificat. Functia returneaza calea completa a fisierului sau mesajul fisier negasit.

**static** String search (File f, String fileName) {

String contents[] = f.list() ;

**int** i = 0;

String found = "fisier negasit" ;

**for** (i=0; i<contents.length; i++) {

**if** (fileName.equals (contents[i]))

**return** (**new** File(f, contents[i]).getAbsolutePath()) ;

}

i = 0 ;

**while** ((i < contents.length) & (found.equals ("fisier negasit"))) {

File child = **new** File (f, contents[i]) ;

**if** (child.isDirectory())

found = *search* (child, fileName);

i++ ;

}

**return** found ;

}

**public** **static** **void** main(String args[]) {

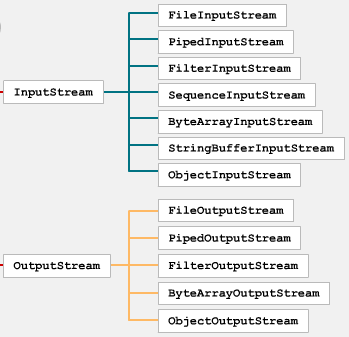
File f= **new** File("E://J2EE");

System.*out*.println(*search*(f,"sampletext.txt"));

}

Functia list() returneaza un sir de stringuri, listand continutul directorului. Daca File nu este un director list() va returna null.

Cand Java citeste sau scrie date, deschide un flux, citeste si scrie informatii in flux si apoi inchide fluxul. Clasele stream utilizeaza intrari si iesiri de date generale in timp ce clasele writer si reader folosesc stringuri Unicode. Datele receptionate de la sau trimise la dispozitive de I/O constau in bytes. Java suporta si alte tipuri de date precum intregi, caracter sau string. Java foloseste o ierarhie de clase pentru a lucra cu diferite tipuri de date. Clasele InputStream si OutputStream sunt abstracte si folosesc fluxuri low-level pentru a citi bytes de la sau a-i trimite la fisiere, sockets sau pipes. Fluxurile low-level au acces direct la bytes procesati. Fluxurile high-level sunt construite peste cele low-level, introducand noi capabilitati.



Clasele FilterInputStream si FilterOutputStream extind clasele InputStream respectiv OutputStream si utilizeaza fluxuri high-level pentru a citi sau scrie date precum stringurile sau intregii din fluxuri de octeti.

Majoritatea claselor de intrare au o clasa de iesire corespunzatoare.

Urmatoarele clase fac operatii de intrare-iesire fara formatare (la nivel de jos, binar):

* ByteArrayInputStream si ByteArrayOutputStream, scriu si citesc siruri de octeti prin zone tampon
* FileInputStream si FileOutputStream, scriu si citesc siruri de octeti din fisiere
* ObjectInputStream si ObjectOutputStream, deserializeaza (serializeaza) date primitive si obiecte
* PipedInputStream si PipedOutputStream, lucreaza cu fire de comunicatie
* SequenceInputStream, permite concatenarea la alte fluxuri de intrare si citirea din fiecare pe rand

Subclasele lui InputStream si OutputStream au structura si functii complementare, incluzand:

* Constructori. Clasele FileInputStream si FileOutputStream au constructori similari. Sintaxa constructorilor este:

FileInputStream(String pathname)  
FileInputStream(File file)  
FileOutputStream(String pathname)  
FileOutputStream(File file)

* Metode de citire si scriere. Metoda read() din FileInputStream citeste urmatorul byte din fisier. Metoda write() din FileOutputStream scrie un byte in fisier. Sintaxa celor doua metode este:

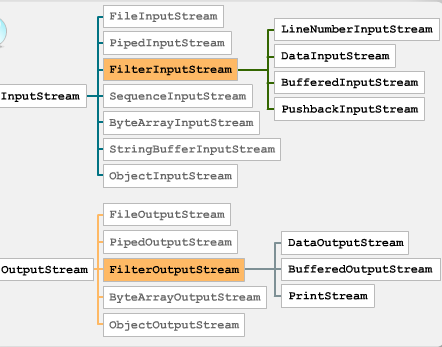
int read () throws IOException   
void write (int <b>) throws IOException

* Metode de citire si scriere a sirurilor. Clasele FileInputStream si FileOutputStream au metode complementare pentru citirea si scrierea sirurilor de octeti. Sintaxa acestor metode este:

int read(byte[] b)   
int read(byte[] b, int off, int len)   
  
void write(byte[] b)   
void write(byte[] b, int off, int len)

Fluxurile de intrare-iesire high-level comunica cu fluxurile low-level.

Majoritatea claselor de intrare iesire de nivel inalt mostenesc atribute din superclasele FilterInputStream si FilterOutputStream. Constructorul unei astfel de clase primeste ca argument un Input/OutputStream. O diagrama a claselor este ilustrata in figura urmatoare:



Presupunand cazul constructorului clasei DataInputStream, va trebui sa-i trimitem ca parametru un constructor InputStream:

DataInputStream(InputStream objectName)

Observatie: Putem folosi oricare dintre clasele care extind clasa abstracta InputStream ca parametru al constructorului.

Cand un obiect flux de nivel inalt receptioneaza octeti de la un flux de nivel jos, proceseaza octetii si ii converteste intr-un tip de date potrivit. Spre exemplu, clasa DataInputStream contine metoda de citire ce converteste bytes catre toate tipurile primitive precum si la stringuri. Metoda readInt() citeste urmatorii patru octeti si ii converteste intr-un intreg. Programatorul este cel care trebuie sa se asigure ca urmatorii patru bytes reprezinta un intreg. Pentru a inchide un flux se utilizeaza metoda close().

Daca dorim sa inchidem un lant de obiecte flux atunci trebuie sa o facem in ordine inversa crearii. Aceasta previne inchiderea unui flux low-level inaintea unui flux high-level.

Dam, in continuare, un exemplu de scriere intr-un fisier, silviu.dat, a unor date in format UTF folosind fluxul high-level DataInputStream construit pe baza fluxului low-level FileOutputStream, care la randul sau s-a creat din fisierul File. De remarcat ca inainte de scrierea prima oara silviu.dat exista ca un fisier vid, in directorul aplicatiei.

**public** **static** **void** main(String args[]) {

File f = **new** File ("silviu.dat");

**if**(f.exists() && f.isFile() && f.canWrite()) {

**try** {

FileOutputStream fostream = **new** FileOutputStream(f);

DataOutputStream dostream = **new** DataOutputStream(fostream);

dostream.writeUTF("date codificate UTF");

dostream.close();

fostream.close();

}

**catch** (IOException e) {

}

}

}

De observat, la sfarsitul codului, modalitatea de inchidere a fluxurilor, in ordine inversa fata de ordinea crearii.

**Clasele Reader si Writer**

Sunt clase abstracte si sunt similare claselor de flux respective. Ambele sunt impartite in clase low-level si high-level. Clasele low-level comunica cu dispozitivele de intrare iesire iar cele high-level cu cele low-level. Aceste clase au fost desemnate special pentru a lucra cu caractere Unicode. Astfel, clasele low-level lucreaza cu caractere.

Avem urmatoarele clase low-level:

* FileReader, folosita pentru a citi fluxuri de caractere dintr-un fisier. Folosim aceasta clasa pentru a citi fisiere text
* CharArrayReader, folosita pentru a citi siruri de caractere, sub forma unui flux de caractere. Clasa implementeaza un buffer de caractere
* PipedReader, folosita pentru a citi caractere pintr-un flux pipe
* StringReader, folosita pentru a citi stringuri

Clasele high-level sunt:

* BufferedReader, folosita pentru a citi texte de la un flux de intrare, de caractere. Este folosita pentru a spori eficienta codului, furnizand un buffer. Este recomandat sa folosim intotdeauna intrari si iesiri folosind un buffer
* FilterReader, este abstracta si este folosita pentru a citi fluxuri de caractere filtrate, dintr-un fisier. Clasele derivate manipuleaza sirurile de caractere citite oferind o filtrare a datelor citite. Spre exemplu, putem filtra liniile citite dintr-un fisier pe baza unei expresii regulate
* InputStreamReader, folosita pentru a converti un stream de bytes intr-o multime de caractere folosind un Charset specificat. Putem folosi aceasta clasa pentru a accepta System.in

Clasa FileReader are doi constructori:

FileReader(String pathname)  
FileReader(File file)

Acesta este formatul general al constructorilor tuturor claselor din ierarhia ce implementeaza direct Reader.

Vom da, in continuare, un exemplu in care vom citi un text de la tastatura, pe care apoi il vom afisa:

**public** **static** **void** main(String args[]) {

**try** {

InputStreamReader buff = **new** InputStreamReader(System.*in*) ;

BufferedReader theFile = **new** BufferedReader(buff);

System.*out*.println("introduceti un text:");

String s = theFile.readLine();

theFile.close();

buff.close();

System.*out*.println("textul introdus este: " + s);

}

**catch**(Exception e){

}

}

Constructia este asemanatoare celei din exemplul precedent.

Majoritatea claselor de citire au in corespondenta clase de scriere.

Daca metoda de citire cel mai frecvent utilizata este readLine(), corespondentul sau la scriere este write(). Scrierea din buffer in destinatie nu se face daca cantitatea de date este mai mica decat cea a bufferului. Abia cand dimensiunea bufferului a fost atinsa se produce scrierea. Pentru a preveni pierderea de informatii datorate inchiderii bufferului folosim metoda flush(). Apelul acesteia determina golirea bufferului si trimiterea tuturor datelor fisierului cu care este in corespondenta. flush() va fi apelata inainte de close().

Cateva dintre exceptiile importante atunci cand lucram cu operatii de I/O sunt:

* FileNotFoundException, localizarea unui fisier se face fara succes
* EOFException, sfarsitul fisierului atins intr-un mod neasteptat
* IterruptedIOException, intrerupere neasteptata
* ObjectStreamException, clasa de baza pentru erorile aruncate de clasele ObjectStream

In Java 6 clasa File contine metode noi ce furnizeaza informatii despre utilizarea discului de memorie:

* getTotalSpace(), returneaza dimensiunea in octeti a partitiei in care se afla fisierul
* getFreeSpace(), returneza dimensiunea in octeti a zonei libere din partitia in care se afla fisierul
* getUsableSpace(), asemanatoare functiei precedente, dar verifica si restrictiile sistemului de operare si permisiunile de scriere asociate fisierului.

Alte extensii din versiunea 6 sunt utilizate pentru a restrictiona sau permite citirea, scrierea sau executarea unui fisier:

* setWritable(), specifica permisiune de scriere asupra unui obiect File. Are doua semnaturi:

setWritable(boolean writable)  
setWritable(boolean writable, boolean ownerOnly)

prima este utilizata pentru a specifica daca fisierul poate fi scris sau nu. A doua specifica daca permisiunea la scriere este valabila doar proprietarului fisierului

* setReadable(), specifica permisiune de citire asupra unui obiect File. Are doua semnaturi:

setReadable(boolean readable)  
setReadable(boolean readable, boolean ownerOnly)

prima este utilizata pentru a specifica daca fisierul poate fi citit sau nu. A doua specifica daca permisiunea la citire este valabila doar proprietarului fisierului. Daca sistemul nu poate decide proprietarul va acorda permisiune tuturor indiferent de ceea ce stabileste aceasta metoda

* setExecutable(), specifica permisiune de executie asupra unui obiect File. Are doua semnaturi:

setExecutable(boolean writable)  
setExecutable(boolean writable, boolean ownerOnly)

Metoda care verifica posibilitate ca un fisier sa poata fi executat este canExecute().

**Pachetul java.nio**

Denumirea provine de la „new I/O”. Inlocuieste java.io si are scopul de a facilita operatii de intrare-iesire neblocabile, dar si blocarea partiala a unui fisier.

NIO are performante mai bune si este mai eficient. Spre exemplu, permite citirea si scrierea folosind o singura conexiune in doua sensuri FileChannel. Permite lucrul in conjuctie cu APIul existent. Principalele facilitati sunt:

* Fisiere mapate si blocate: fisierele mapate in memorie creaza o mapare care ne permite sa lucram cu fisierele de date ca si cum ar ocupa un spatiu in memorie. Fisierele blocate permit accesul exclusiv al unei singure aplicatii sau accesul mai multora
* I/O asincrone: ne permite sa adaugam I/O nonblocabile si selectabile in codul Java. Cand folosim I/O neblocabile toate actiunile necesare pentru citire sau scriere sunt executate imediat. Cand folosim I/O asincrone firul principal este gata sa selecteze un flux care este pregatit sa transmita sau sa receptioneze date. El se muta la un alt flux daca cel original se blocheaza
* Canale: un canal functioneaza ca un punct final pentru comunicatie. Ele seamna cu fluxurile. Dispozitivele folosesc metodele proprii pentru citirea sau scrierea de date. Interfata Channel are doar doua metode: isOpen() si close()
* Bufferi: pachetul este construit in jurul obiectelor ByteBuffer, in timp ce java.io foloseste siruri de octeti. Obiectele ByteBuffer ne permit sa setam bufferele la read-only si sa urmarim pozitia datelor scrise sau citite dintr-un buffer. Ne permit, de asemenea, sa scriem toate tipurile de date primitive. Putem seta bufferii ca bufferi directi, ce utilizeaza bufferi gazduiti de sistemul de operare
* Coderi si decodari de caractere: permit conversia sirurilor de date in caractere si invers. Clasa Charset din acest pachet permite acest lucru.

Buffer este o subclasa directa a lui Object. Buffer inmagazineaza propria stare. Este abstracta. Are trei proprietati:

* position: determina pozitia unde urmatorul element va fi citit sau scris in buffer.
* limit: determina pozitia primului element din buffer ce nu va fi scris sau citit. Limita poate fi intre 0 si capacitatea buferului si poate fi setata la rulare prin apelul metodei limit()
* capacity: este capacitatea bufferului sau numarul sau de elemente. Capacitatea este setata la crearea bufferului

Clasele Buffer si ByteBuffer (clasa care implementeaza interfata) au urmatorele metode:

* clear(), seteaza proprietatea position la zero si limit la valoarea capacitatii. Nu afecteaza datele din buffer. Este folosita atunci cand dorim sa introducem date noi in buffer. Datele initiale se pastreaza dar vor fi suprascrise
* mark(), seteaza o pozitie in buffer la care ne putem intoarce la un apel viitor. Se poate folosi pentru a retine locul in care datele au fost scrise
* reset(), seteaza proprietatea position la pozitia data de mark. Nu afecteaza datele din buffer. Este folositoare atunci cand dorim sa copiem sau sa rescriem date dupa ce au fost sterse
* put(), este folosita pentru a scrie octeti intr-un ByteBuffer. Pozitia in care scriem este intotdeauna intre mark si limit.
* get(), este folosita pentru a citi octeti dintr-un ByteBuffer. Pozitia din care citim este intotdeauna intre mark si limit
* flip(), seteaza limit la position. Apoi seteaza position la 0. Este folosita dupa scrierea de date in buffer
* compact(), muta toate datele la inceputul bufferului. Apoi seteaza position la valoarea unde datele se incheie si limit la valoarea capacitatii

In exemplul urmator vom da construirea si gestionarea unui buffer, folosind metodele anterior descrise:

**import** java.nio.\*;

**public** **class** Test{

**public** **static** **void** showBufferProperties(Buffer b){

System.*out*.println("proprietatile bufferului:");

System.*out*.println("capacitatea="+ b.capacity());

System.*out*.println("limita="+ b.limit());

System.*out*.println("pozitia="+ b.position());

}

**public** **static** **void** showBufferData(ByteBuffer b){

System.*out*.println("afisarea datelor bufferului");

**while**(b.hasRemaining())

System.*out*.println(b.get());

}

**public** **static** **void** showArrayData(**byte**[] array){

System.*out*.println("afisarea datelor bufferului");

**for**(**int** cnt = 0;cnt < array.length; cnt++){

System.*out*.println(array[cnt]);

}

}

**public** **static** **void** main(String[] args){

//construirea unui buffer dintr-un array

System.*out*.println("crearea, popularea si afisarea unui buufer de bytes");

**byte**[] array = {0,1,2,3,4,5,6,7};

*showArrayData*(array);

System.*out*.println("crearea bufferului");

ByteBuffer b = ByteBuffer.*wrap*(array);

*showBufferProperties*(b);

*showBufferData*(b);

*showBufferProperties*(b);

System.*out*.println("modificarea primului element in array");

array[0] = 10;

*showArrayData*(array);

System.*out*.println("apelul met flip() in buffer");

b.flip();

*showBufferProperties*(b);

*showBufferData*(b);

System.*out*.println("rearanjarea bufferului");

b.rewind();

*showBufferProperties*(b);

*showBufferData*(b);

b.put(3,(**byte**)20);

b.rewind();

*showBufferData*(b);

*showArrayData*(array);

System.*out*.println("marcarea la pozitia 2 ");

b.rewind().position(2).mark();

System.*out*.println("setarea pozitiei la 4");

b.position(4);

*showBufferData*(b);

System.*out*.println("resetarea la marcajul anterior");

b.reset();

*showBufferData*(b);

System.*out*.println("Bufferul este read only:"+ b.isReadOnly());

}

}

Canalele sunt folosite pentru a comunica cu fisiere si dispozitive. Canalele sunt cuprinse in pachetul java.nio.

Canalele implementeaza interfata Channel. Aceasta interfata este accesata de cate ori canalul este deschis sau inchis si are metoda close() pentru a inchide canalul. O data inchis un canal nu mai poate fi redeschis.

Avantajele folosirii canalelor:

* o ierarhie a claselor mult mai clara
* operatii de citire scriere comune: un canal reprezinta o conexiune in doua sensuri catre un fisier, socket sau dispozitiv. Putem face operatii de citire scriere folosind o singura conexiune
* imbunatatirea integrarii bufferului: cu ajutorul canalelor putem mapa direct fisierele bufferelor de memorie si atunci putem citi si scrie direct din obiectele ByteBuffer

java.nio.channels cuprinde diverse clase ce implementeaza o varietate de interfete:

* clasa FileChannel implementeaza printre altele interfetele ByteChannel, GatheringByteChannel si ScatteringByteChannel. Cu ajutorul acestei clase putem citi sau scrie intr-un fisier. Permite, de asemenea, maparea unei largi portiuni din fisier memoriei. Este asemanatoare ca functionlitate lui RandomAccessFile.
* clasa SocketChannel implementeaza printre altele interfetele ByteChannel, GatheringByteChannel si ScatteringByteChannel. Permite comunicarea prin socket neblocabil, inchidere asincrona si este thread safe.
* clasa Pipe.SourceChannel implementeaza interfetele ReadableByteChannel si ScatteringByteChannel. Acest canal reprezinta partea de sfarsit ce poate fi citit intr-un pipe.
* clasa Pipe.SinkChannel implementeaza interfetele WritableByteChannel si GatheringByteChannel. Acest canal reprezinta partea de sfarsit ce poate fi scrisa intr-un pipe.

Clasa FileChannel prezinta urmatoarele avantaje ale folosirii ei:

* blocarea fisierului pentru a preveni scrierea sau citirea simultana. Avem doua tipuri de blocare: share-uita si exclusiva. Blocarea exclusiva previne plasarea unei alte blocari atunci cand fisierul este deja blocat
* mapari in memorie pentru cazurile in care dorim ca fisierul sa fie disponibil unui singur obiect ByteBuffer. Fisierul este mapat in memorie ceea ce economiseste resursele atunci cand intentionam sa accesam fisierul
* optimizarea transferului de fisier prin metodele transferTo() si transferFrom() ce pot fi mai eficiente decat un ciclu ce ce copiaza octetii cititi intr-un fisier destinatie

Dam, in continuare, cateva exemple de folosire a facilitatilor acestei clase:

**private** FileChannel myReadWrite ;

**public** FileWriting (String openFile) {

**try** {

// deschiderea unui FileOutputStream in mod append

myReadWrite = **new** FileOutputStream(openFile, **true**).getChannel() ;

}

**catch** (IOException e) {

myReadWrite = **null** ;

}

}

In exemplul anterior am creat un FileChannel dintr-un obiect FileOutputStream prin apelul metodei getChannel(). In mod append, dat de valoarea true al celui de-al doilea parametru al constructorului, variabila position va avea valoarea dimensiunii fisierului.

**public** **void** memoryMapAFile (String fileToMap ) {

// cream un FileChannel si mapam un buffer pentru el

**try** {

FileChannel mappedFile = **new** RandomAccessFile (fileToMap, "rw").getChannel() ;

MappedByteBuffer mappedBuff = mappedFile.map (FileChannel.MapMode.*READ\_WRITE*, 0, mappedFile.size() ) ;

}

**catch** (IOException e) {}

}

In exemplul anterior am folosit functia map() pentru a crea un MappedByteBuffer, care incorporeaza toate metodele din ByteBuffer plus cele referitoare la mapare. Canalul a fost deschis pentru citire si scriere (FileChannel.MapMode.READ\_WRITE).

**public** **void** bulkCopy (String fileIn, String fileOut ) {

**try** {

FileInputStream inFile = **new** FileInputStream (fileIn) ;

FileOutputStream outFile = **new** FileOutputStream (fileOut, **true**) ;

FileChannel in = inFile.getChannel () ;

FileChannel out = outFile.getChannel () ;

in.transferTo (0, (**int**) in.size(), out ) ;

in.close () ;

out.close () ;

}

**catch** (IOException e) {}

}

In exemplul anterior am creat o copie a unui fisier in Java. Metoda transferTo() transfera datele intre cele doua canale. Ca parametru am trimis si dimensiuniea fisierului din canalul sursa. (in.size()).

Metoda wrap(), impacheteaza un sir de octeti intr-un ByteBuffer si il scrie intr-un ByteBuffer.

myBuff = ByteBuffer.wrap(myBytes) ;  
f.myReadWrite.write (myBuff) ;

Metoda force() este folosita pentru ca schimbarile facute asupra continutului unui buffer sa fie scrise si in canal.

System.out.println ( "Bytes written: " + totalBytes ) ;  
 System.out.println ( "File length now: " + f.myReadWrite.size () ) ;  
 f.myReadWrite.force( true ) ;  
 f.myReadWrite.close() ;

Metoda close() determina inchiderea canalului.

**Clasa Scanner**

Permite citirea facila, parsarea stringurilor si tipurilor primitive folosind expresii regulate. Se afla in pachetul java.util. Clasa Scanner citeste intrari de la consola, fisier, string sau orice alte surse ce implementeza interfata Readable sau ReadableByteChannel.

Simplifica citirea datelor formatate, de intrare, cu ajutorul tokenilor ce urmaresc expresii regulate. Un token este o secventa de caractere pe care o specificam atunci cand cream un scanner folosind delimitatori potriviti. In mod predefinit delimitatorul este spatiul.

O expresie regulata este un string ce cuprinde un sablon de potrivire. Modul in care expresia regulata se potriveste unui token in codul sursa va determina parsarea datelor. Putem folosi metode predefinite pentru a extrage expresii precum stringuri, double sau intregi.

Putem sa ne definim si propriile expresii folosind metoda next(). Metoda are ca parametru un Pattern sau un String si returneza un String, care reprezinta urmatorul token ce se potriveste sablonului. Procesul de scanare implica urmatorii pasi:

* scannerul localizeaza un token folosind delimitatorii specifici
* scannerul identifica potrivirea unui token cu o expresie regulata, daca este specificata vreuna
* tokenul este stocat intr-o variabila

Scanner are urmatorii constructori folositi pentru a crea obiecte de citire a intrarilor:

* Scanner(InputStream), dintr-un flux de intrare
* Scanner(InputStream sursa, String charsetName), dintr-un flux de intrare formatat, cel de-al doilea parametru specifica tipul de codificare folosit pentru a converti fluxul de intrare la caractere
* Scanner(String), dintr-un string
* Scanner(File), dintr-un fisier
* Scanner(File sursa, String charsetName), dintr-un fisier formatat, cel de-al doilea parametru specifica tipul de codificare folosit pentru a converti continutul fisierului la caractere
* Scanner(Readable), dintr-un obiect Readable
* Scanner(ReadableByteChannel), dintr-un obiect ReadableByteChannel
* Scanner(ReadableByteChannel sursa, String charsetName), dintr-un obiect ReadableByteChannel, cel de-al doilea parametru specifica tipul de codificare folosit pentru a converti canalul de octeti la caractere

Urmatorul exemplu creaza un scanner de la fluxul standard de intrare.

import java.util.\*;  
  
public class ScanConsole {  
 public static void main(String[] args) {  
 // Declaram si initializam un Scanner de la System.in  
 Scanner in = new Scanner(System.in);  
  
 }  
}

Iar urmatorul dintr-un FileReader

import java.util.\*;  
import java.io.\*;  
  
public class ScanConsole {  
 public static void main(String[] args) {  
  
 FileReader fileIn = new FileReader( "Scanner.txt" );  
 Scanner file = new Scanner(fileIn);  
  
 }  
}

Pentru a citi din fluxul de intrare vom urma pasii:

* determinam daca mai exista ceva de citit folosind metoda public boolean hasNextX(), care este supraincarcata. Printre cele mai utilizate: hasNextBoolean(), hasNextInt()si hasNext(String), unde parametrul este o expresie regulata
* citim efectiv folosind metoda public String nextX(), care este supraincarcata. Printre cele mai utilizate: public boolean nextBoolean(),public int nextInt()si public String next(String), unde parametrul este o expresie regulata. Daca nu exista niciun token se va arunca o NoSuchElementException. Metoda blocheaza scannerul pe timpul citirii intrarii. Pe timpul blocarii metoda se opreste si asteapta ca utilizatorul sa introduca date.
* repetam ultimii doi pasi

Un exemplu de folosire a clasei Scanner este prezentat in cele ce urmeaza:

**public** **static** **void** main(String[] args) {

Scanner in = **new** Scanner(System.*in*);

String str = " ";

System.*out*.println("introduceti numele si varsta sub forma nume varsta,");

System.*out*.println("si apasati enter");

// scaneaza intarea si o blocheaza daca este cazul

**if** (in.hasNext())

str = "numele este " + in.next();

// scaneaza intarea si o blocheaza daca este cazul

**if** (in.hasNextInt())

str += " si varsta este " + in.nextInt();

// afiseaza rezultatul

System.*out*.println(str);

}

Pentru a modifica delimitatorul predefinit folosim metoda Scanner useDelimiter(String/Pattern).

Pentru a determina delimitatorul curent folosim Pattern delimiter().

Spre exemplu apelul metodei:

in.useDelimiter("; \*");

determina setarea delimitatorului la ; urmat de spatiu si apoi orice caractere.

**Iesiri formatate**

Incepand cu 5.0 se introduce metoda printf(). Functia poseda control asupra iesirilor formatate. Functia scrie un string intr-un string de iesire folosind un format specificat si argumente. Functia se foloseste cu fluxuri print, spre ex System.out sau System.err. Putem utiliza metoda cu doua tipuri de obiecte:

* PrintStream, pentru o iesire pe octeti
* PrintWriter, pentru o iesire pe caracter

Cele doua forme ale metodei sunt:

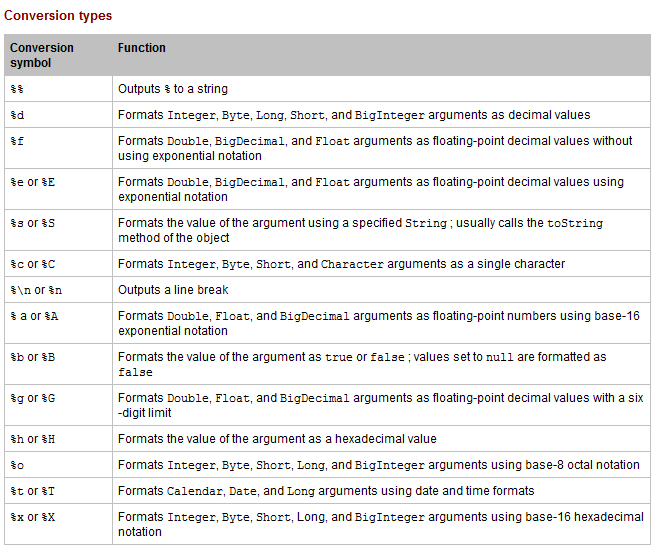
public PrintStream printf(Locale, String, Object);

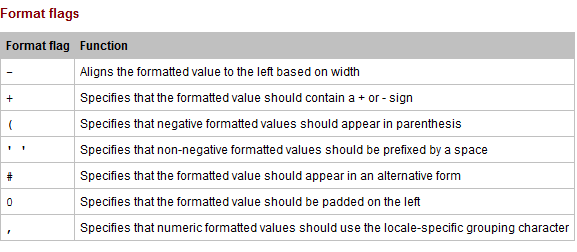
in cazul in care se precizeaza o localizare, respectiv:

public PrintStream printf(String, Object);

ce foloseste o localizare predefinita. Stringul da formatul de care se va tine cont la afisare. Ultimele argumente corespund formatului dat in string. Putem folosi urmatoarele argumente de format:

* %n, inserarea unui sfarsit de linie la afisare
* %e sau %E, afisarea cu exponent a numerelor zecimale. Argumentul trebuie sa fie de tip: Double, BigDecimal sau Float





Vom da, in continuare, un exemplu de folosire a citirii cu formatare:

**public** **static** **void** main(String[] args) {

Date datePrinted = Calendar.*getInstance*().getTime();

// printeaza data completa

System.*out*.printf("data este: %tc\n", datePrinted);

// printeaza un string si un intreg

System.*out*.printf("%s are %d de ani\n", "ion", 30);

// printeaza un double sau un float

System.*out*.printf("greutatea este de: %3.2f kg\n", 150.537);

// printeaza un double folosind formatul stiintific

System.*out*.printf("inaltimea este de: %e cm\n", 180.28);

}

**Clasa Formatter**

Ne permite sa convertim stringuri, numere si date in aproape orice format. Ne permite, de asemenea, sa formatam tipuri de date Java: byte, Calendar si BigDecimal. Se afla in java.util. Clasa Formatter este stricta, ea afiseaza o eroare daca sesizeaza o formatare invalida.

Exemplu de folosire a acestei clase;

**public** **static** **void** main(String[] args) {

**int** numVisitor = 100;

StringBuilder buffer = **new** StringBuilder();

// Construim un obiect Formatter cu localizarea US

Formatter formatter = **new** Formatter(buffer, Locale.*US*);

// Formatam un caracter, un string si un zecimal

formatter.format("%c. sunteti %s %d\n", 'A', "vizitatorul", numVisitor);

//format() face acelasi lucru ca printf()

System.*out*.printf("sunteti %s %d\n", "vizitatorul", numVisitor);

System.*out*.println(formatter);

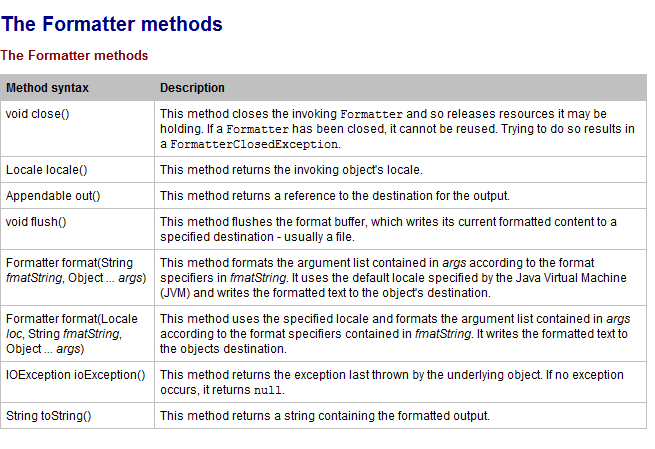
}

In exemplul anterior format() si printf() au acelasi rol.

Clasa Formatter converteste intrari binare in text formatat, care va fi retinut intr-un buffer. Putem obtine continutul bufferului invocand metoda out(). Putem, de asemenea, converti bufferul la un string apeland toString(). Clasa are urmatorii constructori frecvent utilizati:

* Formatter(), creaza un obiect general ce utilizeaza un obiect StringBuilder pentru a stoca iesirea formatata, la locatia predefinita, specificata de JVM
* Formatter(Appendable), creaza un obiect ce stocheaza iesirea formatata intr-un buffer
* Formatter(String) throws FileNotFoundException, creaza un obiect pentru a stoca iesirea formatata intr-un fisier. Daca fisierul nu poate fi creat arunca o exceptie

In plus constructorii clasei pot specifica: locatia obiectului creat, setul de caractere folosit. Cateva dintre metodele clasei:



Metoda format() este folosita pentru a formata argumentele in concordanta cu specificatorii. In modul cel mai utilizat are doi parametri: un string ce contine o combinatie de text si specificatori de format. Specificatorii de format sunt precedati de % urmati de un singur caracter. Pentru fiecare specificator de format trebuie sa fie un obiect in lista de argumente.

Folosim %t si %T ca specificatori de format pentru a afisa date, cu argumente de tip Calendar, Date, Long sau long. Urmatoarele sufixe sunt cele mai utilizate impreuna cu specificatorii de mai sus:

* d, data curenta a lunii, formata din 2 cifre (01-31)
* r, afiseaza timpul in 12 ore (hh:mm:ss AM sau PM)
* c, afiseaza timpul in format hh:mm:ss zona an
* m, afiseaza luna, formata din 2 cifre (01-12)
* M, minutul orei, format din 2 cifre (00-59)
* b, abrevierea lunii (Jan-Dec)
* B, luna completa (Janury-December)
* I, ora curenta

Pentru numere putem specifica lungimea si precizia. Lungimea minima a reprezentarii unui numar se face dupa % indicand valoarea, iar precizia urmeaza lungimii, separata de aceasta prin punct.

**public** **static** **void** main(String[] args) {

**int** numVisitor = 100;

**double** d1 = 9.837;

StringBuilder buffer = **new** StringBuilder();

Formatter formatter = **new** Formatter(buffer, Locale.*US*);

// Formatul numeric octal

formatter.format("formatul octal pentru %d este %o\n", numVisitor, numVisitor);

// Formatul numeric hexadecimal

formatter.format("formatul hexazecimal %f este %a\n", d1, d1);

System.*out*.println(formatter);

}

Urmatoarele flag-uri dupa procent modifica formatul:

* 0, in loc de spatii va adauga zerouri nesemnificative
* +, adauga semnul + in fata numerelor pozitive
* (, scrie numerele negative intre paranteze rotunde
* Spatiu, scrie un spatiu in fata numerelor pozitive pentru alinierea cu numerele negative
* -, aliniere la stanga
* ,, pentru separarea miilor

**public** **static** **void** main(String[] args) {

Date date = **new** Date();

Calendar cal = Calendar.*getInstance*();

StringBuilder buffer = **new** StringBuilder();

Formatter formatter = **new** Formatter(buffer, Locale.*US*);

// afiseaza timpul in format orar de 12 ore

formatter.format("%tr\n", date);

// afiseaza informatii complete despre data/timp

formatter.format("%tc\n", cal);

// afiseaza numai orele/minutele

formatter.format("%1$tl:%1$tM\n", cal);

// afiseaza numele lunii

formatter.format("%1$tB %1$tb %1$tm\n", cal);

System.*out*.println(formatter);

}

Formatarea numerelor:

**public** **static** **void** main(String[] args) {

StringBuilder buffer = **new** StringBuilder();

Formatter formatter = **new** Formatter(buffer, Locale.*US*);

buffer.setLength(0);

*printNums*(formatter);

}

**static** **void** printNums(Formatter formatter) {

**double** d = -2.2834;

**for** (**short** i=1; i <= 8; i++, d += d+i) {

formatter.format("%15f %15.5f %15.2f%n", d, d\*d, d\*d\*d);

}

System.*out*.println(formatter);

}