

## 8. EXCEPTII. LUCRUL CU SISTEMUL DE FISIERE. SERIALIZARE

8.1. Tratarea erorilor unei aplicatii folosind exceptii.....	<a href="#">2</a>
8.1.1. Probleme in managementul erorilor.....	<a href="#">2</a>
8.1.2. Sistemul de exceptii: principii si avantaje.....	<a href="#">4</a>
8.1.3. Aruncarea unei exceptii.....	<a href="#">5</a>
8.1.3.1. Cine si cum arunca exceptii?.....	<a href="#">5</a>
8.1.3.2. Ce tipuri de obiecte pot fi “aruncate”?.....	<a href="#">5</a>
8.1.4. Obiectul exceptie si capabilitatile sale.....	<a href="#">6</a>
8.1.5. Propagarea exceptiilor.....	<a href="#">7</a>
8.1.6. Prinderea si tratarea unei exceptii.....	<a href="#">7</a>
8.1.6.1. Constructia try...catch.....	<a href="#">7</a>
8.1.6.2. Cazul exceptiilor inrudite.....	<a href="#">9</a>
8.1.6.3. Tratarea identica a mai multor exceptii distincte.....	<a href="#">10</a>
8.1.6.4. Blocul finally.....	<a href="#">10</a>
8.1.6.5. Inchiderea automata a resurselor externe.....	<a href="#">11</a>
8.1.7. Exceptii “checked”.....	<a href="#">11</a>
8.1.8. Exceptii definite de catre programator.....	<a href="#">12</a>
8.1.9. Exceptiile si constructorul.....	<a href="#">12</a>
8.2. Lucrul cu sistemul de fisiere.....	<a href="#">13</a>
8.2.1. Principii, categorii de operatii si solutii disponibile.....	<a href="#">13</a>
8.2.2. Operatii de baza cu sistemul de fisiere.....	<a href="#">13</a>
8.2.2.1. Particularitati ale sistemelor de fisiere si independenta de platforma.....	<a href="#">13</a>
8.2.2.2. Solutia clasica: clasa File.....	<a href="#">14</a>
8.2.2.3. Solutia noua: pachetul java.nio.file.....	<a href="#">15</a>
8.2.2.3.1. Clase disponibile si rolul acestora.....	<a href="#">15</a>
8.2.2.3.2. Operatii asupra unei cai: clasele Path si Paths.....	<a href="#">15</a>
8.2.2.3.3. Operatii asupra fisierelor si directoarelor: clasa Files.....	<a href="#">16</a>
8.2.2.3. Accesarea continutului fisierelor.....	<a href="#">16</a>
8.2.3.1. Clasa java.io.RandomAccessFile.....	<a href="#">16</a>
8.2.3.2. Lucrul cu stream-uri.....	<a href="#">17</a>
8.2.3.2.1. Principii generale si tipuri de stream-uri.....	<a href="#">17</a>
8.2.3.2.2. Ierarhiiile de clase stream.....	<a href="#">18</a>
8.2.3.2.3. API-ul claselor stream. Inlantuirea stream-urilor.....	<a href="#">18</a>
8.2.3.4. Selectia fisierelor/directoarelor in aplicatiile cu interfata grafica.....	<a href="#">19</a>
8.3. Serializare: salvarea starii obiectelor.....	<a href="#">20</a>
8.4. BIBLIOGRAFIE.....	<a href="#">21</a>
8.5. Anexa 1 - corespondente java.io - java.nio.file (cf. Java tutorial).....	<a href="#">21</a>
8.6. Anexa 2 (informativa): Preferences API.....	<a href="#">22</a>

## 8.1. Tratarea erorilor unei aplicatii folosind exceptii

### 8.1.1. Probleme in managementul erorilor

In orice aplicatie ce depaseste o anumita anvergura apare importanta problema a gestionarii erorilor. Daca, intr-o aplicatie simpla ce interactiona cu utilizatorul folosind consola, era suficient sa afisam pe ecran un mesaj in momentul detectarii unei situatii de eroare, pentru o aplicatie cu interfata grafica consola nu mai reprezinta o optiune: afisarea mesajelor pe ecran trebuie efectuata prin intermediul componentelor grafice, care nu sunt accesibile decat din cadrul catorva clase ce le contin, in schimb erorile pot aparea in codul din orice alta clasa. In acest ultim caz este necesar "transportul" informatiei de eroare dintr-un punct in altul al programului - eroarea apare in afara unei clase GUI, insa o putem semnaliza utilizatorului numai din cadrul unei astfel de clase. La modul general, ne dorim sa fim informati ori de cate ori apar erori, dar sa avem flexibilitatea de a le trata acolo unde ne este convenabil, si aceasta presupune propagarea informatiei de eroare.

Rularea unui program presupune o succesiune deapeluri de metode, in general imbricate: spre exemplu, metoda *main()* apeleaza metoda *f()* a unui obiect, aceasta la randul sau apeleaza o metoda *g()* a altui obiect s.a.m.d. astfel incat, in orice moment, exista un anumit numar de metode "incepute" dar care nu s-au incheiat deoarece executia a patrunsi de fiecare data mai in adancime, intr-o alta metoda. Succesiunea deapeluri de metode a caror executie a inceput dar nu s-a incheiat inca este numita "**call stack**" (stiva deapeluri de metode) si reprezinta un concept necesar in logica tratarii erorilor. In exemplul anterior, cand executia ajunge in *g()*, call stack-ul se va compune din *main() → f() → g()* in aceasta ordine; odata ce *g()* se incheie, executia se intoarce in *f()* si, la terminarea acestuia, in *main()*, urmand ca la iesirea din *main()* sa se incheie executia programului. Dupa cum vom vedea, propagarea informatiei de eroare urmareste call stack-ul.

Pentru o mai buna intrelegere, sa consideram urmatorul exemplu: intr-o aplicatie cu interfata grafica, la click pe un buton se incarca dintr-un fisier o lista de persoane si se calculeaza pe baza ei diferite statistici. Fiecare persoana are nume, telefon, email, data nasterii etc. Sa presupunem ca aplicatia contine doua clase: clasa *GUI* ce constituie interfata grafica (inclusiv codul de tratare a evenimentelor) si clasa *Stats* ce contine codul pentru efectuarea statisticilor dorite.

Succesiunea de metode apelate la apasarea pe buton este urmatoarea:

- clasa *GUI*:
  - **void actionPerformed()** - metoda executata ca urmare a apasarii butonului si care apeleaza metoda de calcul al statisticilor
- clasa *Stats*:
  - **int calculeazaStatistici(String cale)** - metoda apelata de *actionPerformed()*, care apeleaza la randul sau metoda de incarcare din fisier si apoi opereaza pe datele obtinute
  - **int incarcaDateDinFisier(String cale)** - metoda apelata de *calculeazaStatistici()* pentru a prelua din fisier detaliiile persoanelor

Toate metodele din clasa *Stats* pot intampina situatii de eroare: spre exemplu, calea catre fisier poate fi invalida, persoanele pot avea date eronate etc. Toate aceste scenarii fac ca metoda curenta (cea care a determinat aparitia erorii) sa nu mai poata continua. Dorim sa semnalam utilizatorului aceste situatii prin afisarea cate unei ferestre de dialog informative.

Suntem constienti de urmatoarele situatii de avarie ce pot surveni in executia metodelor amintite:

- metoda *incarcaDateDinFisier()*: fisierul poate fi inexistent; permisiunile acestuia pot sa impiedice masina virtuala sa-i citeasca continutul; continutul sau poate fi invalid (nu contine o lista de persoane salvata anterior)
- metoda *calculeazaStatistici()*: numarul de inregistrari obtinute din fisier poate fi zero; inregistrarile pot contine date invalide (situatii pe care dorim sa le semnalam separat utilizatorului: telefon invalid, mail invalid etc)

***Nota:** in realitate ar mai putea exista si alte situatii, insa ne propunem doar explicarea unor concepte, nu o lista exhaustiva de erori posibile.*

Ne dorim ca aceste informatii sa fie propagate inapoi pe call stack pana in metoda *actionPerformed()* - singurul loc in care avem acces la componentelete interfetei grafice si deci putem comunica cu utilizatorul. Mai exact, orice eroare aparuta trebuie sa aiba doua efecte:

- anularea executiei restului de cod din metoda (aflat dedesubtul liniei ce a cauzat eroarea). Spre exemplu, in metoda *calculStatistici()*, daca incarcarea fisierului esueaza, nu trebuie sa se mai execute codul de calcul al statisticilor
- propagarea informatiei de eroare pana in locul in care ea poate fi afisata utilizatorului. Altfel spus, metoda *actionPerformed()* trebuie sa “afle” de problemele intamplate in adancime ca sa le poata afisa

Pentru a asigura aceste cerinte am putea gandi urmatorul sistem: metodele implicate vor returna valori intregi care indica, in caz de eroare, natura problemei aparute. Atribuim fiecarei erori posibile cate un numar, si alocam fiecarei metode cate un interval numeric, astfel incat sa nu existe suprapunerile intre codurile returnate de diversele metode:

- metoda *incarcaDateDinFisier()* va returna numarul de inregistrari incarcate (un intreg pozitiv), sau o valoare negativa in caz de eroare. Alocam intervalul **-1...-10** erorilor acestei metode
- metoda *calculeazaStatistici()* va returna 0 pentru a indica succesul operatiei, sau o valoare negativa in intervalul **-11...-19** pentru a indica aparitia unei probleme in cadrul metodei (fiecare numar negativ returnat indica o alta problema)

```
class Stats{
    // citirea din fisier returneaza coduri de eroare -1...-10 pt diversele erori posibile:
    // -1 pentru fisier inexistent, -2 pentru fisier inaccesibil (lipsa permisiuni) etc.
    private int incarcaDateDinFisier(String cale){....}
    // calculul statisticilor produce coduri de eroare -11...-20: -11 pt lista goala, -12 pt
    // mail invalid, -13 pentru telefon invalid etc
    public int calculeazaStatistici(String cale){
        int cod = citesteDateDinFisier();
        if(cod==0) return -11;           // lista de inregistrari goala
        else if (cod<0) return cod;    // propagam codurile metodei citesteDateDinFisier()
        ...urmeaza codul de calcul statistici, care returneaza codurile mentionate -11...-20
    }
    ...alii membri ai clasei - ex: cei in care se memoreaza datele citite si statisticile...
}

// metoda de tratare a apasarii butonului din clasa GUI
public void actionPerformed(ActionEvent e){
    int cod = s.calculeazaStatistici();      // s este un obiect de tip Stats
    if(cod<0){
        switch(cod){
            case -1:   // afisare eroare fisier inexistent
            case -2:   // afisare eroare fisier inaccesibil
            case -11:  // afisare eroare lista de persoane goala
            case -12:  // afisare eroare mail invalid
            case -13:  // afisare eroare telefon invalid
            .....etc.....
        }
    }
}
```

Observam modul in care se realizeaza propagarea informatiei de eroare: metoda *calculeazaStatistici()* verifica valoarea produsa de metoda *incarcaDateDinFisier()* si, in cazul unei erori, returneaza mai departe codul de eroare catre metoda *actionPerformed()*. Aceasta analizeaza codul de eroare primit si, in cadrul unui *switch*, afiseaza utilizatorului mesaje de eroare differentiate.

In utilizarea unui sistem de acest fel am constata mai devreme sau mai tarziu urmatoarele neajunsuri:

- **propagarea manuala a informatiei de eroare** - cu urmatoarele dezavantaje:
  - face codul mai greu de urmarit. Fiecare apel de metoda potential generatoare de erori trebuie sa fie inglobat intr-un *if/switch* care sa testeze valoarea de intoarcere si sa ia masuri in caz de probleme; astfel codul devine impanat cu sectiuni de cod care tin de tratarea erorilor si pierde din claritate
  - trebuie sa avem grija intotdeauna sa propagam si erorile venite “din adancime”. Observam cum metoda *calculeazaStatistici()* preia codurile de eroare generate de *incarcaDateDinFisier()* si le transmite catre *actionPerformed()*
- **poluarea valorii de iesire a metodelor cu informatii care tin de erori** - cu dezavantajele:
  - tipul de date de intoarcere al metodei poate fi de asa natura incat sa nu permita distinctia intre erori. Daca metoda *incarcaDateDinFisier()* ar fi returnat nu numarul de inregistrari, ci insasi lista de persoane (sa spunem, sub forma unui obiect colectie *List*), atunci singurele ei valori de iesire erau fie o referinta la *List*, fie *null*. Putem folosi *null* ca valoare particulara pentru a indica erorile de incarcare din fisier, insa nu vom mai

putea distinge intre ele - codul client al metodei *incarcaDateDinFisier()* (in speta metoda *calculeazaStatistici()*) va afla ca a avut loc o eroare dar nu va sti care a fost aceea.

- uneori tipul de date de intoarcere nici nu ar permite desemnarea unor valori particulare care sa indice eroarea. Spre exemplu, daca metoda *incarcaDateDinFisier()* ar fi returnat nu numarul de inregistrari, ci o informatie numERICA care putea fi si negativa, am fi fost pusii in imposibilitatea de a folosi rezultatul de intoarcere al metodei pentru a-i semnaliza erorile catre codul client
- **codurile de eroare sunt prea putin descriptive.** Chiar si pentru metodele care isi permit sa returneze valori particulare pentru a-si indica erorile, un simplu numar sau o valoare *null* sunt insuficiente pentru informarea codului client sau a utilizatorului. Am dori ca, acolo unde tratam o eroare, sa avem la dispozitie informatii suplimentare; in exemplul nostru, atunci cand o persoana are email sau telefon invalid, am dori sa-i afisam utilizatorului si numele persoanei in cauza, impreuna cu valoarea invalida gasita (poate problema este una simpla si usor remediablea - ex: yahoo.co in loc de yahoo.com!). Aceasta informatie trebuie propagata integral din metoda *calculeazaStatistici()* catre *actionPerformed()*

Remarca: intelegem din cele de mai sus ca erorile apar deseori intr-un loc din program dar trebuie tratate in alt loc; aceasta deoarece, in astfel de cazuri, "vina" aparitiei erorii nu este a metodei care a detectat-o, ci a datelor sale de intrare - asadar a codului client. Este fireasca deci oprirea executiei metodei curente (pentru a proteja restul codului sau, care ar fi afectat de aparitia erorii) si pasarea responsabilitatii catre codul client pentru a trata erorile descoperite.

### 8.1.2. Sistemul de exceptii: principii si avantaje

Sistemul de exceptii adreseaza neajunsurile enumerate mai sus in urmatoarele moduri:

- eroarea nu mai este o simpla valoare particulara, ci devine obiect, putand astfel incapsula orice fel de informatie necesara despre eroarea aparuta
- propagarea exceptiei in sens invers prin call stack este automatizata de catre masina virtuala si nu mai afecteaza cu nimic valorile de intoarcere ale metodelor parcuse

La intalnirea unei erori care face ca executia metodei curente sa nu mai poata continua au loc urmatoarele operatii:

- programatorul creeaza obiectul exceptie, al carui tip este ales de asa natura incat sa descrie cat mai bine situatia de avarie aparuta, si care incapsuleaza toata informatia necesara "la cald". Numim acest pas **generarea exceptiei**
- obiectul este inmanat masinii virtuale (care va hotari apoi ce se va intampla cu el). Numim acest pas **aruncarea exceptiei**
- masina virtuala parcurge in sens invers call stack-ul, cautand un asa-numit handler (o portiune de cod ce trateaza acel tip de eroare). Numim acest pas **propagarea exceptiei**
- primul handler gasit intr-una dintre metodele din call stack este executat ca reactie la aparitia erorii, handlerul extragandu-si informatiile necesare din obiectul exceptie. Numim acest pas **prinderea si tratarea exceptiei**

Iata ce forma capata codul prezentat anterior in conditiile folosirii sistemului de exceptii:

```

class Stats{
    private List incarcaDateDinFisier(String cale){ // putem acum returna colectia de persoane
        if(fisier inexistent) throw new RuntimeException("Fisierul nu exista!");
        if(fisier inaccesibil) throw new RuntimeException("Fisierul nu poate fi citit!");
        ....citire fisier si returnare colectie de inregistrari...
    }
    public void calculeazaStatistici(String cale){
        List persoane = incarcaDateDinFisier(cale);
        if(persoane.size() == 0) throw new RuntimeException("Lista de persoane este goala!");
        //cod de calcul al statisticiilor, care produce si el exceptii in caz de date invalide
    }
    ...altri membri ai clasei...
}
// metoda de tratare a apasarii butonului din clasa GUI
public void actionPerformed(ActionEvent e){
    try{
        s.calculeazaStatistici(); // s este un obiect de tip Stats
    }catch(Exception e){ // prinde toate erorile derivate din Exception, inclusiv Runtime!
        // in ipoteza in care am avea un JLabel numit statusBar pe post de bara de mesaje:
        statusBar.setText(e.getMessage());
    }
}

```

```
// afisarea mesajului de eroare encapsulat in exceptie, indiferent de tipul erorii
}
```

Dupa cum constatam, tratarea erorilor folosind sistemul de exceptii prezinta urmatoarele avantaje:

- cod mai usor de urmarit - se separa codul ce poate genera erori de cel de tratare a erorilor
- propagare automata a informatiei de eroare - nu mai este nevoie ca programatorul sa colecteze si sa propage manual erorile unei metode in cea anterioara din call stack
- crearea de categorii de erori - gratie relatiei de mostenire intre clasele exceptiei, se creeaza grupuri de erori inrudite (spre deosebire de codurile de eroare care erau distincte), si care in acest fel pot fi tratate la comun, pe categorii, in caz de nevoie

### 8.1.3. Aruncarea unei exceptii

#### 8.1.3.1. Cine si cum arunca exceptii?

In Java, o exceptie poate aparea in trei moduri:

- exceptie creata si aruncata de catre programator ca urmare a detectarii unei situatii in care executia nu poate continua in contextul curent. Spre exemplu, daca o metoda trebuie sa imparta doua numere si constata ca impartitorul este 0, nu poate efectua operatia de impartire si in consecinta trebuie sa se inchiele prematur, semnalizand situatia de eroare catre codul sau client (metoda anterioara din call stack) prin aruncarea unei exceptii
- exceptie creata si aruncata de o metoda a unei clase din JRE. Clasele componente ale JRE-ului contin o multitudine de metode care pot arunca exceptii la apelare; spre exemplu, metoda *Integer.parseInt()* arunca o exceptie de tip *NumberFormatException* daca sirul de caractere pasat ca argument nu are un format numeric valid
- exceptie creata si aruncata de catre masina virtuala insasi - astfel de exceptii apar ca urmare a unor situatii de avarie detectate de catre JVM. Exemple: folosirea unui index negativ sau prea mare intr-un tablou (*ArrayIndexOutOfBoundsException*), incercarea de accesare a unui membru de obiect cand referinta are valoarea null (*NullPointerException*) etc.

#### 8.1.3.2. Ce tipuri de obiecte pot fi "aruncate"?

Dupa cum s-a explicat, obiectele exceptie create nu sunt returnate din metode, ci folosesc un mecanism paralel: sunt "inmanate" masinii virtuale care va asigura propagarea lor inapoi prin call stack in cautarea unui potential handler. Crearea unui obiect exceptie si pasarea sa masinii virtuale se numeste in Java "aruncarea unei exceptii" (**exception throwing**) – de unde si instructiunea asociata (**throw**). Iata sintaxa de aruncare a unei exceptii:

```
throw new Exception("Mesaj de eroare demonstrativ");
```

In acest exemplu instructiunea **throw** primeste ca parametru un obiect de tip *Exception*, generat pe loc. Alternativ, obiectul putea fi creat anterior, putea avea alt tip de date, putea sa nu incapsuleze un mesaj etc.

Tipurile de obiecte care pot fi "aruncate" (pasate ca argument lui *throw*) sunt cele derivate din clasa *java.lang.Throwable*. In general vor fi folosite subclase directe sau indirecte ale acesteia - iata descrierea claselor relevante:

- **Throwable** – descendant direct al lui *Object*; este clasa parinte pentru toate erorile si exceptiile din Java
  - **Error** – reprezinta erori fatale ce nu are sens sa fie tratate de catre aplicatie (ex: *StackOverflowError*, *OutOfMemoryError*). Ele pot fi prinse si tratate, insa in general este tardiv: masina virtuala s-a straduit deja sa nu ajunga in situatia respectiva si nu mai exista masuri pe care sa le poata lua programatorul pentru a indrepta situatia
  - **Exception** – indica erori *trataabile* de catre aplicatie; este clasa parinte pentru exceptii.
    - descendantii directi ai clasei *Exception* sunt exceptiile numite "checked" – aceste exceptii pot aparea indiferent de calitatea codului scris de catre programator (ex: o conexiune de retea se poate intrerupe, un fisier e posibil sa fi fost sters intre timp etc). Din acest motiv, compilatorul se

asigura ca programatorul a luat masuri impotriva unor situatii de acest gen, fortandu-l fie sa prinda si sa trateze exceptia (*try...catch*), fie, daca nu, sa o declare in semnatura metodei

- **RuntimeException** – exceptii ce pot fi aruncate in cursul functionarii normale a JVM. Sunt deseori - dar nu obligatoriu - rezultatul unei erori de programare (de exemplu, *ArrayIndexOutOfBoundsException*, *AritheticException* sau *NullPointerException*). Ele nu fac obiectul verificarii la compilare (deoarece in mod normal nu ar trebui sa apara) si deci nu trebuie declarate in semnatura metodei care le genereaza si nici nu este obligatoriu sa fie prinse si tratate

Există o multitudine de tipuri derivate din *Exception* sau *RuntimeException*, distribuite in diversele pachete Java. Nu vom gasi exceptiile grupate la un loc (cum ar fi un pachet *java.exception*); ele sunt definite in aceleasi pachete cu clasele care le genereaza. De exemplu, *IOException* este definita in *java.io*, *AWTException* in *java.awt* etc. Lista de exceptii definite in fiecare pachet poate fi gasita in documentatia Java API, in fereastra ce contine lista claselor din pachetul respectiv (frame-ul din stanga jos). Pentru fiecare metoda care poate arunca exceptii este indicata lista exacta in documentatia sa.

Iata cateva exemple de exceptii mai des intalnite si cauzele lor:

- **IOException** – erori aparute in operatiunile de input/output (ex: citirea dintr-un fisier sau din retea). O subclasa de interes este **FileNotFoundException**, generata in cazul incercarii de utilizare a unui fisier inexistent
- **NullPointerException** – folosirea unei referinte cu valoarea *null* pentru accesarea unui membru de obiect
- **ArrayIndexOutOfBoundsException** – folosirea unui index ilegal (negativ sau prea mare) intr-un tablou
- **AritheticException** – operatiuni aritmetice ilegale (ex: impartire cu 0)
- **IllegalArgumentException** – indica pasarea unui argument incorrect intr-o metoda. O subclasa de interes este **NumberFormatException** ce corespunde erorilor de transformare din String in primitiva in metodele *parse\**() ale claselor de impachetare
- **ClassCastException** – apare la conversia unei referinte la un alt tip de date, incompatibil
- **SQLException** - erori aparute la interogarea serverelor de baze de date

#### 8.1.4. Obiectul exceptie si capabilitatile sale

Un obiect exceptie este generat atunci cand, in urma unei erori, executia metodei curente nu mai poate fi continuata si este necesara revenirea la metoda anterioara din call stack. Obiectul incapsuleaza date despre eroarea aparuta (tipul de eroare, un mesaj ce poate fi afisat utilizatorului, stiva de apeluri de metode in momentul aparitiei erorii etc).

Scopul unui obiect exceptie este sa transporte informatie despre eroare de la “fata locului” pana la handler. Cantitatea de informatie necesara depinde de tipul de eroare: daca un utilizator a introdus o varsta invalida intr-un formular, ar putea fi suficiente incapsularea in obiectul exceptie a unui mesaj de eroare care semnaleaza problema; in schimb, in cazul unei erori in interogarea unui server de baze de date, ne dorim o informatie mai bogata - datele de conectare la server, codul si mesajul de eroare generate de server etc. Din acest motiv exista diverse tipuri de clase exceptie, fiecare incapsuland mai multe sau mai putine campuri in functie de caz.

Toate clasele exceptie au elemente comune mostenite de la clasa *Throwable*, dintre care cel mai important este mesajul. Acesta poate fi initializat in constructor si beneficiaza de getter, astfel incat codul handler sa-l poata extrage si, uneori, chiar afisa direct utilizatorului. Metode relevante din clasa *Throwable*:

- **Throwable(String)** - stabileste mesajul incapsulat. Toate sub clasele exceptie ofera si un constructor cu argument de tip *String* care se foloseste de cel parinte pentru a initializa mesajul
- **String getMessage()** - extrage mesajul incapsulat in obiectul exceptie
- **Throwable(Throwable), initCause(Throwable)** - folosite in cazul exceptiilor inlantuite pentru a seta exceptia cauzatoare a unei exceptii
- **getStackTrace() si printStackTrace()** - permit accesul la elementele call stack-ului, asa cum arata el in momentul aparitiei exceptiei

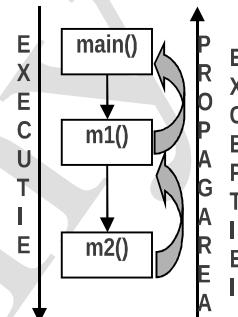
Multe clase exceptie “se multumesc” cu metodele mostenite din *Throwable*, deoarece simplul tip de date pe care il creeaza permite tratarea separata a acelui tip de eroare. Unele dintre ele adauga insa informatie suplimentara in obiectul exceptie; spre exemplu, *SQLException* contine codul si mesajul de eroare generate de serverul de baze de date.

## 8.1.5. Propagarea exceptiilor

Sistemul de exceptii este, in parte, un mecanism de control al executiei, deoarece determina saltul executiei dintr-un loc in altul al programului. La aparitia unei exceptii intr-o metoda, instructiunile aflate dedesubtul liniei cauzatoare de exceptie nu se vor mai executa, ceea ce reprezinta o forma de menajare a codului care ar fi fost afectat de aparitia exceptiei. Spre exemplu, daca la incercarea de introducere a unei noi inregistrari in baza de date esueaza conectarea la server, tot codul ce-i urmeaza in metoda (care efectueaza interogarea si apoi inchiderea conexiunii) este pus in imposibilitatea de a se mai executa si trebuie evitat.

Odata aruncat, obiectul exceptie este preluat de masina virtuala care incepe sa caute un handler pentru acel obiect, dupa cum urmeaza:

- daca a fost prevazut un handler local sub forma unui bloc *try...catch* si exista un bloc *catch* care prinde acel tip de exceptie sau unul mai larg, se executa blocul in cauza si apoi restul de instructiuni ce urmeaza in cadrul metodei. Cu aceasta tratarea exceptiei se incheia (vezi detalii in sectiunea despre prinderea unei exceptii)
- daca nu exista un handler local, masina virtuala se intoarce in metoda anterioara din call stack, cautand acolo un handler (spunem ca exceptia a fost *propagata*). De aici incolo lucrurile decurg dupa acelasi tipic: daca in acea metoda este gasit un handler acesta va trata exceptia, in caz contrar executia se muta cu o metoda in urma pe call stack s.a.m.d. Daca exceptia nu este tratata la niciunul dintre nivelurile call stack-ului ea ajunge in final in metoda terminala unde determina incheierea firului curent de executie, cu afisarea in consola a stack trace-ului



**Nota:** programatorul are si optiunea de a prinde o exceptie si de a arunca o alta pe baza ei.

Iata un exemplu care ilustreaza manifestarea nativa a unei exceptii ne tratate:

```
class Exceptie {
    void metoda(int i){
        if(i<0) throw new RuntimeException("i<0!!");
    }
    void f() { metoda(-1); };
    public static void main(String[] args) {
        Exceptie e=new Exceptie();
        e.f();
        System.out.println("Mesaj post-exceptie");
    }
}
```

Rezultatul rularii acestui program este:

```
Exception in thread "main" java.lang.RuntimeException: i<0!!
at Exceptie.metoda(Exceptie.java:4)
at Exceptie.f(Exceptie.java:6)
at Exceptie.main(Exceptie.java:9)
```

Se vede cum exceptia apare in *metoda()*, dupa care, nefiind tratata local, ajunge in *f()* (de unde a fost apelata *metoda()*) si apoi in *main()* a carui executie se incheie cu afisarea stack trace-ului. Mesajul post-exceptie nu va mai fi afisat.

## 8.1.6. Prinderea si tratarea unei exceptii

### 8.1.6.1. Constructia try...catch

In propagarea sa inapoi prin call stack, o exceptie poate fi interceptata si tratata: se numeste ca programatorul a definit un **handler** pentru ea (o portiune de cod care se “ocupa” de ea). Exceptia poate fi interceptata la oricare dintre niveluri, incepand cu cel in care a aparut si terminand cu metoda *main()*<sup>1</sup>; desigur in cazul unei erori nu putem lua masuri in contextul curent de executie (metoda sau blocul in care a fost detectata eroarea) si in aceste cazuri este preferabila tratarea

<sup>1</sup> Aceasta pentru o aplicatie obisnuita; applet-urile nu au metoda *main()*, ele disponand de alte metode, apelate de catre browser

exceptiei la nivelurile superioare. **Daca o exceptie este interceptata si tratata ("prinsa") pe unul dintre niveluri, ea nu se mai propaga in continuare.**

In scopul tratarii exceptiei, programatorul delimita zonele de cod „periculoase” (cele care pot genera exceptii) folosind un bloc **try**; acesta este urmat de unul sau mai multe blocuri **catch** ce au rolul de handler, iar la sfarsitul intregii constructii poate fi atasat un bloc optional **finally**.

```
try{
    // instructiuni ce pot arunca exceptii
}catch(TipExceptie1 e){
    // handler pentru TipExceptie1
}catch(TipExceptie2 e){
    // handler pentru TipExceptie2
}
...alte blocuri catch...
finally{      // bloc optional!
    // cod executat indiferent daca a aparut sau nu exceptie
}
// cod ce urmeaza constructiei try...catch
```

**Nota:** nu este permisa introducerea de cod intre blocurile constructiei **try..catch..finally** (ex: *instructiuni intre try si catch sau intre catch si finally*)

Blocurile **catch** sunt un fel de metode-reactie rulate la aparitia unui anumit tip de exceptie; argumentul primit este insusi obiectul exceptiei, din care codul de tratare a acesteia isi poate extrage datele necesare.

Evolutia executiei este urmatoarea:

- daca in blocul **try** nu apare nicio exceptie, instructiunile acestuia se vor executa pana la final si apoi se va trece la instructiunile ce urmeaza constructiei **try...catch**
- daca in blocul **try** apare o exceptie, acesta se incheie prematur - instructiunile blocului aflate dedesubtul celei “vinovate” nu se mai executa. Masina virtuala preia obiectul exceptie si cauta un handler pentru el:
  - daca exista un bloc **catch** atasat care prinde fie acel tip exact de exceptie, fie unul parinte, instructiunile acelui bloc **catch** vor fi rulate pe post de handler. Daca handler-ul nu arunca la randul sau o exceptie, executia va continua cu instructiunile de dupa constructia **try...catch**
  - daca nu exista un bloc **catch** care prinde acea exceptie, masina virtuala se va comporta ca in cazul oricarei exceptii ne tratate: va incheia prematur executia metodei si va incepe sa caute un handler in metodelle anterioare din call stack

Observatii:

- o exceptie aruncata dintr-un **catch** nu poate fi prinsa de catre unul dintre celelalte **catch-uri**. Blocurile **catch** nu prind decat exceptiile aruncate in blocul **try** atasat
- blocurile **catch** se excluz reciproc: nu exista nicio situatie in care “se arunca mai multe exceptii” si deci s-ar executa mai multe blocuri **catch** cu ocazia unei unice situatii de eroare. Chiar daca instructiunile din blocul **try** pot arunca mai multe exceptii, nu vor face niciodata acest lucru simultan - orice exceptie aparuta duce la neexecutarea restului de cod din **try** si deci imposibilitatea aparitiei unei alte exceptii

```
// scrierea intr-un fisier a unui intreg cu valoarea 13
try {
    FileOutputStream fos = new FileOutputStream(new File("c:\\\\temp\\\\test.txt"));
    fos.write(13);                                // 1
    fos.close();                                 // 2
    Object o = fos; String s = (String)o;          // 3
} catch (FileNotFoundException ex) {
    System.out.println("Fisierul dorit nu exista");
} catch (IOException ex) {
    System.out.println("Nu s-a putut scrie in fisier");
}
```

**Atentie!** Orice variabila definita intr-un bloc **try** sau **catch** este locala acelui bloc, nefiind vizibila in afara lui!

In acest exemplu, instructiunea 1 poate genera *FileNotFoundException*, instructiunile 2 si 3 - *IOException*, iar linia 4 - *ClassCastException*. Succesiunea operatiilor este urmatoarea: daca instructiunea marcata 1 genereaza o exceptie, executia blocului *try* se va opri in acel punct (liniile 2-4 nu se mai executa dupa aparitia acesteia!) si se cauta un handler pentru *FileNotFoundException*-ul aparut. Aceasta este gasit in „persoana” blocului *catch* corespunzator, ale carui instructiuni vor fi executate in continuare. Daca liniile 2 sau 3 produc un *IOException*, va fi folosit blocul *catch* corespunzator. In schimb, daca linia 4 produce *ClassCastException*, exceptia nu beneficiaza de un handler local si va fi pasata metodei de la nivelul superior, unde cautarea unui handler va continua.

Observam cum au fost grupate mai multe instructiuni generatoare de exceptii in blocul *try*, atasand apoi acestuia cate un bloc *catch* pentru fiecare tip de exceptie ce se doreste a fi tratata distinct. In acest fel se realizeaza o buna separare intre logica aplicatiei si logica de tratare a erorilor.

**Nota:** se observa ca constructia *try...catch* este un dispecer de exceptii, realizand un fel de switch(TipExceptie) si directionand astfel fiecare exceptie catre blocul de cod ce o trateaza; de aceea insusi numele clasei exceptie este prima si poate cea mai relevanta caracteristica a exceptiei, caci el ofera posibilitatea executarii de cod distinct pentru tipuri de situatii de eroare diferite. Acesta este si motivul pentru care multe dintre clasele exceptie predefinite nici nu au nevoie sa adauge campuri si metode suplimentare fata de cele din *Throwable*; acelasi motiv poate determina programatorul sa-si scrie propriile clase exceptie - pentru a putea trata diferit anumite situatii de eroare specifice.

**Atentie!** Unii programatori incepatori sunt agasati de necesitatea prinderii si tratarii erorilor si cad prada tentatiei de a folosi urmatorul procedeu:

```
try{
    // cod care arunca exceptii
}catch(Exception e){}
```

Acest procedeu (bloc *catch* cu corp vid) poarta numele de *inghitirea exceptiei* (exception swallowing) si NU este recomandat, deoarece anuleaza insusi scopul sistemului de exceptii: anuntarea unei erori si propagarea informatiilor legate de aceasta. O exceptie trebuie prinsa numai daca o putem trata la acel nivel; in caz contrar ea trebuie lasata sa se propage pentru a informa nivelurile anterioare din call stack sau, in ultima instanta, utilizatorul.

### 8.1.6.2. Cazul exceptiilor intrudite

Atunci cand intr-o singura constructie **try...catch** sunt generate mai multe tipuri de exceptii, ele pot fi tratate independent folosind mai multe blocuri *catch*, insa nu oricum:

- daca intre tipurile de exceptii generate nu exista nici o relatie de mostenire, putem folosi cate un bloc *catch* pentru fiecare tip de exceptie, in orice ordine dorim
- daca unele tipuri de exceptii sunt derive din altele, ordinea blocurilor *catch* trebuie sa fie de la particular catre general (prima data vor fi tratate tipurile derive si abia apoi tipurile parinte). Aceasta datorita faptului ca, prin upcasting, un bloc *catch* care are ca parametru tipul de exceptie parinte va „prinde” si tipurile de exceptii derive din acesta, in virtutea relatiei “is-a”:

```
// pseudocod
try {
    // cod care poate genera RuntimeException, IndexOutOfBoundsException si
    // StringIndexOutOfBoundsException
}
// catch (RuntimeException re){...}// nu il punem aici, deoarece celelalte doua tipuri de exceptii
// sunt derive din RuntimeException si ar fi tratate tot aici
catch(StringIndexOutOfBoundsException e) {...} // cel mai particular primul
catch(IndexOutOfBoundsException e){...} // parintele lui StringIndexOutOfBoundsException
catch(RuntimeException e){...} // cel mai general tip dintre cele generate in blocul try
```

Relatiile de rudenie intre exceptii ofera un avantaj important: posibilitatea prinderii cu un singur *catch* a unei intregi ierarhii de exceptii. Spre exemplu, *IOException* prinde implicit si *FileNotFoundException*, *IllegalArgumentException* prinde si *NumberFormatException*, *RuntimeException* prinde toate exceptiile runtime iar *Exception* prinde toate exceptiile! Ne putem folosi de aceasta proprietate in doua moduri:

- pentru a trata identic o intreaga familie de exceptii (ex: un *catch(IOException)* executa acelasi cod pentru toate exceptiile subclaselor sale)
- pentru a fi siguri ca nu “ne scapa” exceptii, utilizand un bloc final *catch(Exception)*:

```
try{ ...cod... }
catch(TipExceptie1 e){...}
catch(TipExceptie2 e){...}
...
catch(Exception e){ ... cod care afiseaza sau/si introduce in loguri eroarea... }
```

### 8.1.6.3. Tratarea identica a mai multor exceptii distincte

Sectiunea anterioara a aratat cum putem proceda pentru a trata identic exceptii inrudite; cum procedam insa pentru exceptii distincte? O prima solutie este sa atasam blocului *try* care produce exceptiile mai multe blocuri *catch* - cate unul pentru fiecare tip de exceptie - ce vor contine cod identic; solutia nu este insa nici eleganta, nici eficiente.

Incepand cu Java SE 7 exista posibilitatea ca un singur bloc *catch* sa prinda/trateze mai multe tipuri de exceptii. Sintaxa presupune specificarea tipului de date al exceptiei sub forma unei liste de tipuri separate prin caracterul '|' (bara verticala):

```
try{ ... cod ... }
catch(IOException|SQLException e){ ... cod tratare ... }
```

Acest procedeu introduce doua restrictii:

- exceptiile incluse in lista nu pot fi inrudite (ex: *catch(Exception|RuntimeException)*)
- atunci cand un bloc *catch* trateaza mai multe tipuri de exceptii, argumentul (referinta la obiectul exceptie) este implicit *final* si deci nu poate fi modificat din cadrul blocului.

### 8.1.6.4. Blocul finally

Optional, constructia *try...catch* poate fi urmata de un bloc **finally**. Blocul **finally** nu are parametri. El fost gandit sa ofere garantia executarii de cod la finalul constructiei *try...catch* indiferent daca a aparut sau nu o exceptie; scopul este eliberarea de resurse - spre exemplu, fisiere sau conexiuni de retea deschise si care trebuie inchise odata ce zona “periculoasa” din *try* a fost depasita intr-un fel sau altul.

Blocul **finally** va fi executat intotdeauna dupa blocurile *try* si *catch*, dupa cum urmeaza:

- daca in *try* nu apare exceptie, *finally* este executat imediat dupa *try*
- daca in *try* este aruncata o exceptie:
  - daca exista un bloc *catch* corespunzator, acesta va fi rulat dupa intreruperea executiei lui *try*, urmat de *finally*
  - daca nu exista bloc *catch*, se executa *finally* imediat dupa *try* si abia apoi se paraseste metoda curenta, cautand handler in cea anterioara din call stack

```
FileOutputStream fos = null;
ObjectOutputStream oos = null;
try{
    fos = new FileOutputStream(new File("c:\\\\temp\\\\f1.txt"));
    oos = new ObjectOutputStream(fos);
    oos.writeObject(object1);
} catch(FileNotFoundException e){
    System.out.println("Fisier inexistent");
} finally{
    if(oos!=null) oos.close();
    if(fos!=null) fos.close();
}
```

Cele doua fluxuri de date (controlate prin intermediul referintelor *fos* si *oos*) trebuie inchise odata ce nu mai avem nevoie de ele. In blocul *try*, oricare dintre cele trei linii poate produce o exceptie; exceptia *FileNotFoundException* este prinsa,

insa cea de tip *IOException* nu. Pentru a avea garantia inchiderii fluxurilor de date indiferent de situatie, plasam instructiunile de inchidere in blocul finally, executat chiar si in cazul exceptiei neprinse *IOException*.

**Nota:** finally se executa chiar si atunci cand folosim return in cadrul blocurilor try sau catch!

```
void f() {
    try { System.out.println("Inainte de return"); return; }
    finally { System.out.println("Finally"); } // mesajul se va afisa!
}
```

Singura situatie in care *finally* nu e executat este atunci cand in cadrul blocului *try* se apeleaza *System.exit()*.

### 8.1.6.5. Inchiderea automata a resurselor externe

O solutie alternativa pentru garantarea inchiderii resurselor externe este oferita incepand cu Java 7, sub forma constructiei denumite **try-with-resources**. Aceasta presupune includerea deschiderii fluxurilor de date ca argument al blocului *try*; efectul este inchiderea automata a acestora fara interventia programatorului, indiferent de aparitia sau nu a unei exceptii:

```
try(ObjectOutputStream fos = new ObjectOutputStream(new FileOutputStream(new File("c:\\f1.txt"))){
    oos.writeObject(obiect1);
}catch(FileNotFoundException e){
    // cod de tratare
}
```

Pot fi folosite pe post de resurse atasate blocului *try* orice fel de obiecte care implementeaza *java.lang.AutoCloseable* (si care dispun, in consecinta, de o metoda *close()* ce le va putea fi astfel apelata automat).

### 8.1.7. Exceptii "checked"

Exceptiile *checked* sunt cele derive din *Exception* care insa nu sunt subclase ale lui *RuntimeException*. Ele corespund erorilor care pot aparea indiferent de calitatea codului nostru (ex: eroare la citirea/scrierea de date in retea, intreruperea comunicatiei cu un server de baze de date sau lipsa privilegiilor necesare etc). Daca celelalte doua categorii de obiecte *Throwable* - erorile si exceptiile runtime - pot fi ignorate fara a afecta procesul de compilare, compilatorul nu ne va permite sa ignoram exceptiile checked, ci ne va obliga sa luam una din doua masuri: fie sa le tratam local, fie sa le declarăm in semnatura metodei in care sunt aruncate. Cele doua posibilitati sunt descrise cu sintagmele "*handle or declare*" sau "*catch or specify*".

**Nota:** exceptiile derive din *RuntimeException* nu sunt de tip checked si nu exista obligativitatea tratarii sau declararii lor.

Cele doua solutii oferite de compilator nu sunt nicidecum echivalente:

- **handle** (sau catch) presupune prinderea si tratarea locala a exceptiei. Aceasta solutie are sens atunci cand putem lua masuri pentru a trata exceptia local (spre exemplu, putem afisa un mesaj catre utilizator). Nu vom prinde o exceptie atunci cand nu putem face nimic in privinta ei!
- **declare** (sau specify) presupune declararea exceptiei in semnatura metodei, ca in exemplul de mai jos:

```
void citire() throws IOException { System.in.read(); }
```

Trebuie intelese ca apelul unei metode care arunca exceptii checked devine la randul sau linie cauzatoare de exceptii checked si deci face subiectul acleiasi impuneri din partea compilatorului:

```
void citesteLinie(){
    citire(); // nu compileaza, trebuie luate aceleasi doua masuri posibile ca mai sus
}
```

Deducem asadar ca declararea unei exceptii in semnatura metodei inseamna de fapt pasarea explicita a responsabilitatii catre codul client al acesteia. Vom proceda astfel numai atunci cand dorim sa fortam codul client sa ia masuri in privinta exceptiei in cauza.

Cand o metoda poate genera mai multe tipuri de exceptii checked, ele vor fi enumerate in declaratia ei, separate prin virgula:

```
void f() throws IOException, SQLException { /* corpul metodei */ }
```

In cazul overriding-ului exista impuneri in privinta setului de exceptii generate de varianta rescrisa a metodei (cea din subclasa). Daca in cazul mostenirii este posibila (si recomandabila) largirea facilitatilor in cadrul subclasei, pentru setul de exceptii al unei metode rescrise lucrurile stau exact invers - el poate fi in cel mai bun caz restrans:

- metoda rescrisa are voie:
  - sa genereze un subset al exceptiilor metodei parinte pe care o rescrie
  - sa genereze exceptii de acelasi tip cu cele din metoda originala, sau tipuri derivate din cele originale
- nu este permis ca metoda rescrisa:
  - sa genereze tipuri suplimentare de exceptii fata de metoda parinte (cu exceptia cazului in care acestea sunt derivate din tipurile metodei parinte)
  - sa arunce exceptii mai generale decat cele din metoda parinte

### 8.1.8. Exceptii definite de catre programator

JRE ofera o intreaga serie de clase exceptie predefinite, care acopera multe dintre situatiile de eroare existente in programare. De aici intrebarea legitima: ce ar determina programatorul sa creeze propriile-i clase exceptie? Motivele pot fi diverse:

- uneori, exceptiile deja existente nu descriu in mod satisfacator o situatie de eroare. *IllegalArgumentException* poate fi o informatie prea vaga, pe cand *VarstaNegativaException* descrie optim situatia si permite (prin simplul sau nume distinct) tratarea separata a *acelei* erori specifice aplicatiei
- obiectele exceptie existente pot sa nu contina informatiile necesare programatorului. Pentru a dicta ce anume informatie este transportata de obiectul exceptie, programatorul poate crea propria clasa exceptie cu ce atribute considera necesare

Programatorul are posibilitatea de a-si defini propriile tipuri de exceptii prin crearea de clase derive din *Throwable* sau din subclasele sale. O subclasa de *Exception* va fi automat o exceptie checked, pe cand una de *RuntimeException* o exceptie unchecked.

```
class VarstaNegativaException extends RuntimeException {
    // stabilirea mesajului incapsulat
    public VarstaNegativaException(int v) { super("Varsta "+v+" nu poate fi negativa!"); }
}
// exceptia poate fi folosita apoi ca oricare alta:
...
try {
    if(varsta<=0) throw new VarstaInvalidaException();
}
catch(VarstaInvalidaException e) { /* cod tratare */ }
```

### 8.1.9. Exceptiile si constructorul

Privit dintr-un anume unghi, constructorul clasei este un fel de setter initial: el permite introducerea de date externe in interiorul obiectului chiar cu ocazia instantierii (aceasta fiind, de altfel, singura ocazie in cazul unei clase care nu beneficiaza de setteri). Ca orice setter, acesta trebuie sa valideze datele inainte ca ele sa fie memorate in obiect; cu mijloacele de pana acum nu aveam posibilitatea de a opri instantierea la detectarea unor date invalide, insa exceptiile ne ofera solutia cautata.

**Nota:** diferența principală între constructor și un setter obisnuit era ca, în cazul celui de-al doilea, aveam optiunea unui simplu return în caz de date invalide, ceea ce ar fi lasat starea curentă a obiectului nealterată, pe cand în primul caz return nu face decât să încheie constructorul prematur, lasând însă campurile obiectului cu valorile lor default.

```
class Persoana{
    private String nume;
    private int varsta;
    public Persoana(String n, int v){
        if(n == null || n.length() == 0)
            throw new IllegalArgumentException("Nume de persoana invalid!");
        if(v<0) throw new VarstaInvalidaException(v);
        nume = n; varsta = v;           // executia ajunge aici numai daca nu s-a aruncat exceptie
    }
}
```

## 8.2. Lucrul cu sistemul de fisiere

### 8.2.1. Principii, categorii de operatii si solutii disponibile

In toate sistemele de operare majore este disponibila o modalitate de stocare persistenta a informatiei sub forma unuia sau mai multor sisteme de fisiere. Desi exista diferente de la o platforma la alta, organizarea informatiei este peste tot una arborescenta (existand conceptul de director sau folder) iar operatiile cu sistemul de fisiere sunt in mare aceleasi. Divizam operatiile posibile in urmatoarele categorii:

- operatii de baza - se refera la creare/stergere/reenumire/aflare informatii despre fisiere si directoare. Java ofera doua solutii:
  - pachetul **java.io** si mai exact clasa **File** - este solutia clasica si mai inflexibila, dar in acelasi timp mai simpla
  - pachetul **java.nio.file** cu clasele sale **Path,Paths,FileSystem,Files** - solutia noua, flexibila dar mai complexa
- accesarea continutului fisierelor - de asemenea cu doua solutii:
  - clasa **RandomAccessFile** - permite accesul liber la continutul unui fisier; poate fi folosita exclusiv pentru fisiere
  - clasele de tip **stream** (flux de date) - reprezinta o solutie mai complexa insa foarte generala, stream-urile putand fi folosite pentru a scrie/citi date si din conexiuni de retea sau alte surse de informatie, nu numai din fisiere

### 8.2.2. Operatii de baza cu sistemul de fisiere

#### 8.2.2.1. Particularitati ale sistemelor de fisiere si independenta de platforma

Sistemele de fisiere intalnite in sistemele de operare moderne prezinta un numar de diferente de care programatorul Java trebuie sa tina cont daca doreste sa scrie aplicatii cu adevarat independente de platforma. Prezentam trei aspecte majore:

- numarul de arbori de resurse distincti in care este organizata informatia:
  - in Windows, corespunzator fiecarei partitii se creeaza implicit un asa-numit drive, care beneficiaza de propria sa litera (C:, D: etc.) si care contine propriul arbore de resurse. Trei partitii inseamna trei arbori diferiti, fiecare cu radacina sa, si deci trei radacini distincte
  - in Linux si Unix, chiar si atunci cand sistemul de fisiere este distribuit pe mai multe partitii, acestea sunt cuplate astfel incat formeaza un singur arbore de resurse. Nu mai exista drive-uri si litere de drive, iar radacina este unica
- simbolul pentru radacina sistemelor de fisiere
  - in Windows acesta este \ (backslash). Exista cate o radacina pentru fiecare drive: C:\, D:\ etc.
  - in Linux simbolul folosit este /, iar radacina este unica
- simbolul care separa doua elemente consecutive ale unei cai: in Windows se foloseste \ (backslash) iar in Linux/Unix caracterul / (forward slash).
- aspectul cailor absolute (consecinta a celor prezentate mai sus):
  - in Windows, o cale absoluta este de forma c:\temp\f1.txt
  - in Linux/Unix, o cale absoluta ia forma /usr/local/src/f1.txt

Diferentele se rezolva prin metode care produc lista de radacini de sisteme de fisiere si prin constante care memoreaza simbolul folosit ca separator in cadrul unei cai. (vezi detalii mai jos, la prezentarea claselor)

### 8.2.2.2. Solutia clasica: clasa File

In ciuda numelui (usor derulant), un obiect de tip *File* incapsuleaza o **cale** in sistemul de fisiere. Aceasta reprezinta un simplu sir de caractere care poate sa corespunda sau nu unui fisier real, nefiind facuta nicio verificare in acest sens la crearea obiectului. Majoritatea claselor din pachetul *java.io* care permit interactiunea cu fisiere desemneaza aceste fisiere folosind obiecte *File*.

Obiectele de tip *File* pot fi create pasand calea ca parametru constructorului:

```
// Windows
File f=new File("c:\\\\java\\\\exemplu");

// Linux/Unix
File f = new File("/java/exemplu");
```

De remarcat dublarea caracterului “\\” in cazul Windows, deoarece acesta este folosit in *String*-uri ca “escape character”(cel care, pus inaintea unui caracter cu semnificatie speciala, ii anuleaza acestuia acea semnificatie; de aceea “\\” are la randul lui semnificatie speciala si trebuie precedat de un escape character “\\”).

Pentru usurarea scrierii de aplicatii independente de platforma, clasa *File* ofera urmatoarele constante:

- **File.separator** (de tip *String*) si **File.separatorChar** (de tip *char*) - au ca valoare separatorul dintre doua elemente consecutive ale caii (\ in Windows si / in Linux)
- **File.pathSeparator** (de tip *String*) si **File.pathSeparatorChar** (de tip *char*) - caracterul folosit pentru separarea mai multor cai dintr-o succesiune ( ; in Windows si : in Linux)

Clasa *File* pune la dispozitie metode de interactiune cu sistemul de fisiere (insa nu si de citire din/scriere in fisier!):

- determinarea radacinilor de sisteme de fisiere disponibile:
  - **File[] File.listRoots()** - produce un element in Linux si cate unul per drive in Windows
- extragerea caii incapsulate si a componentelor sale
  - **String getAbsolutePath()** - produce calea absoluta catre fisier
  - **String getName()** - produce numele fisierului (ultima portiune din calea incapsulata)
  - **String getParent()** - produce calea catre parinte, sau *null* daca nu exista parinte
- verificarea existentei si a tipului de fisier
  - **boolean exists()** – intoarce *true* daca exista o resursa (de orice natura - fisier/director etc) corespunzatoare caii
  - **boolean isFile/isDirectory()** – returneaza *true* daca fisierul este de acel tip
- determinarea permisiunilor pe care le are masina virtuala asupra fisierului
  - **boolean canRead()/canWrite()/canExecute()** - returneaza *boolean* daca masina virtuala are acea permisiune asupra fisierului
- creare si stergere de fisiere si directoare
  - **boolean createNewFile()** - creare de fisier nou, gol; daca fisierul exista returneaza *false*
  - **boolean mkdir()/mkdirs()** - creare de director. Atunci cand argumentul este o cale din care unul sau mai multe directoare intermediare nu exista inca, *mkdirs()* va incerca crearea tuturor directoarelor necesare, pe cand *mkdir()* va esua
  - **delete()** – stergerea nodului pentru care se apeleaza metoda (directoarele trebuie sa fie goale!)
- determinarea dimensiunii unui fisier: **long length()**
- listarea unui director
  - **list()** – produce un tablou de *String* ce contine numele fisierelor din director (doar numele, nu caile complete)
  - **listFiles()** - produce un array de *File* cu fisierile continue in director
- redenumirea unui fisier/director: **boolean renameTo()**

Cu o singura exceptie, toate cele prezentate sunt metode de instanta, ceea ce face operatiile de stergere sau creare a unui fisier sau director usor contra-intuitive: trebuie creat intai obiectul de tip *File*, iar acestuia i se apeleaza metoda in cauza:

```
File f = new File("c:\\\\temp\\\\f1.txt");
System.out.println(f.getName());           // f1.txt
System.out.println(f.getParent());          // c:\\temp
f.createNewFile();
f.delete();
```

## 8.2.2.3. Solutia noua: pachetul java.nio.file

### 8.2.2.3.1. Clase disponibile si rolul acestora

Pachetul *java.nio.file* ofera o intreaga bogatie de clase, in care complexitatea este pe masura flexibilitatii. Ca ghidaj de inceput iata o scurta prezentare a principalelor clase:

- interfata **Path** - un obiect de tip *Path* incapsuleaza o cale, asemanator obiectelor de tip *File*. Spre deosebire de acestea din urma insa, *Path* contine strict metode ce opereaza asupra caii, nu si cele pentru creare/stergere etc de fisiere si directoare
- clasa **Paths** - prezinta o colectie de metode statice (factory methods) ce produc obiecte de tip derivat din *Path*
- clasa **Files** - ansamblu de metode statice ce corespund operatiilor de baza cu fisiere si directoare (creare, stergere, redenumire etc)
- clasa abstracta **FileSystem** - obiectele derive de la aceasta permit interactiunea cu sistemele de fisiere accesibile masinii virtuale
- clasa **FileSystems** - ansamblu de metode statice ce produc obiecte *FileSystem*

### 8.2.2.3.2. Operatii asupra unei cai: clasele Path si Paths

Un obiect de tip *Path* poate fi obtinut prin intermediul metodelor clasei *Paths* dupa cum urmeaza:

- **Path Paths.get(URI)** - produce un obiect *Path* pe baza unuia de tip *URI*
- **Path Paths.get(String cale, String...componente aditionale)** - produce un obiect de tip *Path* corespunzator caii obtinute prin combinarea portiunilor de cale specificate. Cel de-al doilea argument poate lipsi integral; atunci cand este prezent, calea este formata adaugand la primul argument succesiunea de elemente din cele urmatoare, separand fiecare doua elemente consecutive prin caracterul specific platformei:

```
// Windows:
Path p = Paths.get("c:\\\\temp");           // c:\\temp
Path p = paths.get("c:\\\\temp","java","demo"); // c:\\temp\\java\\demo
String[] dirs = {"temp","java","demo"};
Path p = Paths.get("c:\\\\", dirs);           // c:\\temp\\java\\demo
// Linux:
Path p = Paths.get("//","usr","local","src"); // /usr/local/src
```

Un obiect *Path* percepce o cale ca fiind o insiruire de elemente in aceasta ordine:

- un eventual element radacina (ex: *C:\* sau *D:\* in Windows, */* in Linux). Acesta este prezent numai in cazul cailor absolute
- o eventuala insiruire de etichete (directoare) intermediare separate prin caracterul specific al acelui sistem de fisiere. Spre exemplu, pentru calea *c:\\temp\\java\\teste\\*, elementele intermediare sunt *temp* si *java*
- elementul terminal - fisierul sau directorul ce constituie tinta caii (resursa catre care aceasta pointeaza). In cazul caii *c:\\temp\\java\\teste* acesta este *teste*

Metodele clasei *Path* permit accesarea independenta a elementelor (radacina, elemente intermediare, element terminal) si chiar iterarea prin ele (deoarece *Path* extinde *Iterable*):

```
Path cale = Paths.get("/usr/local","src","apache2");
for(Path p: cale){
    System.out.print(p+"\t");           //      usr      local   src     apache2
}
```

Odata obiectul *Path* creat, acestuia i se pot apela metode ce opereaza asupra caii incapsulate:

Categorie	Metode	Descriere
extragere de componente ale caii	String getFileName() Path getName(int pozitie) Path subpath(int pozitieStart, int pozitieStop) Path getRoot() Path getParent()	- produce numele fisierului/directorului tinta (elementul terminal al caii) - produce eticheta aflata pe pozitia specificata (numerotarea se face de la 0) - produce un subset al elementelor caii, cuprins intre pozitiile solicitate - returneaza radacina caii, daca are, sau null in caz contrar - returneaza calea catre directorul parinte al elementului terminal al caii
informatii despre cale	getNameCount()	- returneaza numarul de elemente (etichete) ce compun calea
conversie cale	URI toUri() String toString() Path toAbsolutePath() File toFile()	- creeaza un URI ce incapsuleaza calea din obiectul Path - produce reprezentarea text a caii (calea insasi) - produce calea absoluta tinand cont de directorul curent - produce un obiect de tip File ce incapsuleaza aceeasi cale

```
Path p = Paths.get("C:\\\\temp\\\\java\\\\f1.txt");
System.out.println(p.getNameCount()); // 3
System.out.println(p.getRoot()); // C:\\
System.out.println(p.getFileName()); // f1.txt
System.out.println(p.getName(1)); // java
System.out.println(p.getParent()); // \\temp\\java
```

### 8.2.2.3. Operatii asupra fisierelor si directoarelor: clasa Files

Clasa *Files* reprezinta un ansamblu de metode statice ce corespund principalelor operatii elementare cu sistemul de fisiere: creare, stergere, redenumire, listare etc de fisiere si directoare.

Anumite operatii au nevoie de optiuni suplimentare; in general acestea sunt exprimate prin argumente de tip varargs ale caror valori sunt elemente ale unor enum-uri predefinite. Exemple:

- atunci cand un fisier este link<sup>2</sup> catre altul, trebuie sa putem indica daca dorim accesarea fisierului link insusi sau a tintei sale. In acest scop exista enum-ul *LinkOptions* cu unicul element NOFOLLOW\_LINKS
- atunci cand copiem un fisier exista diverse optiuni pe care ni le putem exprima, cuprinse in enum-ul *StandardCopyOptions*: COPY\_ATTRIBUTES (copierea permisiunilor fisierului impreuna cu continutul sau), REPLACE\_EXISTING (suprascrierea fisierului destinatie daca exista deja) etc.

verificare existenta	boolean exists(Path,LinkOption) boolean notExists(Path,LinkOption)	metodele nu sunt una inversul celeilalte! pentru un fisier inaccesibil nu se poate decide in niciuna dintre directii!
permisiuni	boolean isReadable(Path) boolean isWritable(Path) boolean isExecutable(Path)	verifica daca procesul in care ruleaza masina virtuala are permisiunea in cauza pe fisierul specificat
stergere	void delete(Path)	stergerea unui fisier sau a unui director. Directoarele trebuie sa fie goale!
copiere	copy(Path sursa, Path destinatie, CopyOption...)	la copierea de directoare este copiat doar directorul, fara fisierele continue!
mutare si redenumire	Path move(Path sursa, Path destinatie, CopyOption...)	permite mutarea de directoare numai daca aceasta nu presupune si mutarea continutului (ex: in Unix/Linux, in cadrul aceliasi partitii)
determinare tip fisier	isRegularFile(Path) isDirectory(Path) isSymbolicLink(Path)	verificarea tipului de fisier: fisierele obisnuite sunt cele care contin direct informatie, directoarele sunt folosite pentru organizarea fisierelor, iar link-urile sunt fisiere care nu contin direct informatie, ci pointeaza catre alte fisiere sau directoare
dimensiune	long size(Path)	determinarea dimensiunii fisierului (in octeti)

### 8.2.3. Accesarea continutului fisierelor

#### 8.2.3.1. Clasa java.io.RandomAccessFile

Aceasta clasa este folosita pentru citirea din/scrierea in orice pozitie dintr-un fisier. Este un descendant direct al clasei *Object* si functioneaza independent de alte clase. Constructorul primeste ca parametru o cale (data fie ca *String*, fie sub forma unui obiect *File*) si un mod de acces, care poate fi:

2 In Unix/Linux, un link este un fisier de sine statator care redirectioneaza catre altul (asemanator cu shortcut-ul din Windows)

Studentul poate utiliza prezentul material si informatiile continute in el exclusiv in scopul asimilarii cunoștințelor pe care le include, fără a afecta dreptul de proprietate intelectuală detinut de autor.

- “**r**” – deschiderea fisierului pentru citire
- “**rw**” – citire si scriere. Nu exista posibilitatea deschiderii unui fisier numai pentru scriere

Iata un exemplu:

```
File f=new File("c:\\\\java\\\\fisier1");
RandomAccessFile raf = new RandomAccessFile(f,"r");
```

*RandomAccessFile* priveste fisierul ca pe un sir de octeti accesabili independent. Odata creat, un astfel de obiect dispune de un cursor intern (*file pointer*) care indica pozitia curenta in cadrul fisierului - locul in care va fi efectuata urmatoarea operatiune, fie ea citire, scriere etc. Fiecare citire/scriere de octet avanseaza cursorul cu o pozitie, iar la citirea/scrierea unei primitive avansarea se face cu un numar de pozitii corespunzator. Printre metodele clasei se numara:

- citire din /scriere in fisier
  - **read()** – citeste un octet din fisier (valoare pozitiva in intervalul 0...255!), returnand -1 daca s-a atins sfarsitul fisierului
  - **readByte(),readChar(),readInt(),readDouble()** etc – citeste din fisier o primitiva de tipul specificat (sunt disponibile metode pentru toate tipurile de primitive)
  - **write()** – scrie octetul specificat ca argument la pozitia curenta a cursorului, suprascriind valoarea existenta
  - **writeByte(),writeInt(),writeDouble()** etc – scrierea in fisier a primitivelor specificate (disponibile pentru toate primitivele)
- manipulare cursor
  - **getFilePointer()** – intoarce pozitia cursorului in cadrul fisierului
  - **seek()** – pozitioneaza cursorul la offset-ul specificat fata de inceputul fisierului
  - **skipBytes()** – “sare” numarul specificat de octeti, avansand cursorul
- determinarea lungimii fisierului: **long length()**
- inchiderea fisierului: **close()**

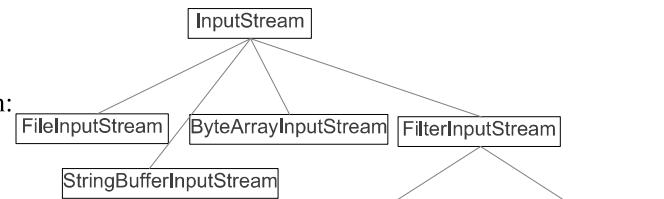
### 8.2.3.2. Lucrul cu stream-uri

#### 8.2.3.2.1. Principii generale si tipuri de stream-uri

Java I/O se bazeaza pe ideea de **stream** (flux de date) – un obiect ce abstractizeaza o sursa/destinatie seriala pentru informatie. Din punct de vedere concret, in spatele unui stream se poate afla un fisier, un sir de octeti, un buffer sau o conexiune in retea. Stream-ul, odata conectat cu dispozitivul aflat in spatele lui, ascunde programatorului detaliiile accesului la date specific fiecarui tip de sursa/destinatie de date in parte.

Stream-urile pot fi clasificate din urmatoarele puncte de vedere:

- din punct de vedere al directiei curgerii datelor, avem:
  - input streams - sunt fluxuri de date prin intermediul caror aplicatia citeste date din surse externe
  - output streams - fluxuri de date ce permit trimitera informatiei din aplicatie catre destinatii externe (fisiere, alte masini virtuale aflate in retea etc)
- din punct de vedere al naturii datelor ce tranziteaza stream-ul, avem:
  - stream-uri orientate pe octet - datele ce le parcurg sunt simpli octeti, iar obiectul stream ne ofera doar posibilitatea de a citi/scrise date octet cu octet
  - stream-uri orientate pe caracter - obiectul stream este constient de faptul ca un caracter poate fi reprezentat sub forma a mai multi octeti si face translatiile necesare acolo unde este cazul, tinand cont de setul de caractere folosit
- din punct de vedere al relatiei cu datele si al independentei fata de alte stream-uri, avem:
  - stream-uri elementare - sunt cele care au acces direct la sursa/destinatia de date. Aceste stream-uri au in general capacitatii foarte reduse (pot lucra doar cu octeti sau cu caractere)



- stream-uri de nivel inalt - sunt fluxuri de date gandite sa interfaseze cu cele elementare pentru a oferi servicii suplimentare (ex: “coagularea” octetilor in primitive sau chiar obiecte). Un astfel de stream nu functioneaza pe cont propriu deoarece nu are acces direct la date; el este gandit sa fie cuplat la un stream low-level

Clasele reale reprezinta combinatii ale tipurilor de mai sus: clasele de tip stream orientat pe octet formeaza doua ierarhii, avand in varf clasele *InputStream* si *OutputStream*; asemanator, stream-urile orientate pe caracter au ca parinti clasele *Reader* (stream-uri de intrare) si *Writer* (stream-uri de iesire). Dedesubtul acestor clase se gasesc atat stream-uri elementare cat si de nivel inalt.

### 8.2.3.2.2. Ierarhile de clase stream

Prezentam in continuare ierarhia fluxurilor de intrare orientate pe octet si caracter. Pentru fluxurile de iesire exista ierarhii asemanatoare (majoritatea claselor au nume deductibil, in care cuvantul *input* este inlocuit cu *output* si *reader* cu *writer*):

- **InputStream** (clasa abstracta) – parintele tuturor claselor de tip flux de date de intrare orientate pe octet; subclasele sale sunt:
  - subclase directe care corespund fluxurilor elementare. Fiecare clasa corespunde cate unei surse posibile de informatie: **ByteArrayInputStream**, **StringBufferInputStream**, **FileInputStream** etc
  - subclase ce implementaza stream-uri de nivel inalt - un fel de „adaptoare” ce se cupleaza la un stream elementar:
    - **FilterInputStream**, cu subclasele:
      - **BufferedInputStream** – “decoreaza” un *InputStream* adaugandu-i un buffer si deci posibilitatea de accesare ne-sequentiala a datelor din stream. Un astfel de flux eficientizeaza lucrul cu datele deoarece sistemul de operare nu mai este nevoie sa faca cate o operatie de citire pentru fiecare apel de metoda a stream-ului; stream-ul citeste un intreg buffer odata si apoi livreaza din el octeti aplicatiei, fara a mai apela la sistemul de operare pana cand bufferul se golest
      - **DataInputStream** – adauga functionalitate unui stream elementar dand posibilitatea aplicatiei de a citi direct primitive
      - **ObjectInputStream** – o sursa de date folosita pentru citirea de obiecte salvate anterior
- **Reader** - clasa parinte a tuturor stream-urilor de intrare orientate pe caracter. Exemple de subclase utile:
  - subclase care implementeaza stream-uri elementare: **CharArrayReader**, **StringReader**
  - subclase care implementeaza stream-uri de nivel inalt:
    - **InputStreamReader** - un stream care se cupleaza la unul de octeti si furnizeaza caractere (puntea intre cele doua “lumi”). Cea mai importanta subclasa a sa este **FileReader** care citeste caractere dintr-un fisier
    - **BufferedReader** - flux de date ce beneficiaza de un buffer, oferind, de exemplu, posibilitatea citirii linie cu linie

### 8.2.3.2.3. API-ul claselor stream. Inlantuirea stream-urilor

Clasele de tip stream prezinta un API care se imbogateste pe masura ce creste nivelul stream-ului: daca stream-urile elementare ofera doar posibilitatea lucrului cu octeti, cele de nivel inalt agrega octetii formand primitive sau chiar obiecte. Exemplificam in continuare API-ul catorva stream-uri de intrare, cu mentionarea ca majoritatea metodelor au corespondent in stream-urile de iesire:

Clasa	Exemple de metode	Descriere/particularitati
InputStream	int read() int read(byte[] b) void skip(long) void close()	- citeste urmatorul octet din stream - citeste un numar de octeti egal cu dimensiunea tabloului (sau mai mic, daca nu mai exista date) - “sare” numarul de octeti specificat din stream. In cazul unui fisier, avanseaza in cadrul acestuia - inchide stream-ul
DataInputStream	readInt(), readFloat() etc	metode de citire de primitive (disponibile pentru toate primitivele)
Reader	int read() int read(char[] b) void skip(long) void close()	- citeste urmatorul caracter din stream si il intoarce sub forma de valoare int pozitiva - citeste urmatoarele caractere din stream in tabloul primit ca argument - “sare” numarul de caractere specificat - inchide stream-ul
BufferedReader	readLine()	- citeste urmatoarea succesiune de caractere pana la intalnirea unuia dintre caracterele \n, \t sau \r

Dupa cum s-a explicat, un stream de nivel inalt nu poate functiona in lipsa unui element care sa ii asigure accesul la date. Acest lucru devine evident inca din constructorul stream-ului de nivel inalt, care solicita un argument de tip elementar. Sa luam exemplul citirii dintr-un fisier. Mai intai trebuie definit obiectul de tip *File* ce reprezinta calea catre fisierul nostru; apoi, pe baza lui, cream un *FileInputStream* ce va oferi citire seriala (byte-cu-byte) din fisier; pe baza lui cream un obiect de tip *DataInputStream* cu ajutorul caruia putem citi direct primitive. *DataInputStream* accepta ca parametru in constructor orice obiect de tip *InputStream* (si implicit oricare din clasele descendente), asadar putem crea un *DataInputStream* direct dintr-un obiect *FileInputStream*:

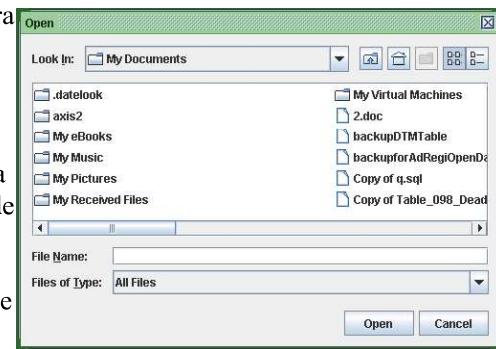
```
File f=new File("c:\\java\\fisier.bin");
FileInputStream fis=new FileInputStream(f);
DataInputStream dis=new DataInputStream(fis);
// acum putem citi din fisier primitive
char c = dis.readChar();
byte b = dis.readByte();
double d = dis.readDouble();
dis.close(); fis.close(); // inchiderea ambelor stream-uri
```

## 8.2.4. Selectia fisierelor/directoarelor in aplicatiile cu interfata grafica

In aplicatiile grafice este deseori nevoie ca utilizatorul sa aleaga un fisier pe care doreste sa-l utilizeze ca sursa de date sau pentru salvarea acestora (ex: celebrele optiuni Open... sau Save... din meniul File al multor aplicatii). Aceasta presupune posibilitatea de a naviga prin sistemele de fisiere disponibile si a selecta fisierul/directorul dorit. In acest scop JRE ofera clasa *JFileChooser*, ale carei obiecte reprezinta ferestre de dialog ca in figura urmatoare.

Iata cateva dintre metodele de interes ale clasei *JFileChooser*:

- **showOpenDialog(Component parinte)** si **showSaveDialog(Component parinte)** - afiseaza pe ecran fereastra de dialog, setandu-i fereastra parinte specificata. Fereastra va include un buton Cancel si unul Open sau Save (in functie de metoda apelata). Executia programului va trece mai de apelul catre aceasta metoda numai dupa ce utilizatorul inchide fereastra. Metoda produce un rezultat care indica in ce fel a inchis utilizatorul fereastra, sub forma unei constante a clasei *JFileChooser*: APPROVE\_OPTION (utilizatorul a apasat butonul de Open sau Save), CANCEL\_OPTION (butonul de cancel) sau ERROR\_OPTION (fereastra este inchisa prin alte mijloace (ex: click pe icon-ul de inchidere) sau are loc o eroare)
- **void setCurrentDirectory(File)** si **File getCurrentDirectory()** - folosite pentru a seta sau extrage directorul curent afisat de catre fereastra de dialog
- **void setFileSelectionMode(int)** - stabileste tipul de fisiere care pot fi selectate, sub forma unor constante din clasa *JFileChooser*: FILES\_ONLY, DIRECTORIES\_ONLY sau FILES\_AND\_DIRECTORIES
- **File getSelectedFile()** - extrage calea catre elementul selectat. **Atentie! Nu presupuneti niciodata ca acesta este un fisier existent! Fereastra ii permite utilizatorul sa editeze manual calea catre fisierul dorit si numele acestuia!**
- **void setMultiSelectionEnabled(boolean)** - stabileste daca fereastra va permite selectarea simultana a mai multor elemente din sistemul de fisiere
- **File[] getSelectedFiles()** - folosit pentru a extrage lista de resurse selectate atunci cand este permisa selectia multipla



```
class GUI extends JFrame{
    // la apasarea pe Open... din meniul File al aplicatiei se deschide fereastra de dialog
    public void openMenuItemActionPerformed(ActionEvent e){
        if(jfc.showOpenDialog(this) == JFileChooser.APPROVE_OPTION){
            File fisierSelectat = jfc.getSelectedFile();
            // ...apoi prelucrare fisier etc...
        }
    }
}
```

## 8.3. Serializare: salvarea starii obiectelor

Java ofera o facilitate numita *lightweight persistence*, adica posibilitatea memorarii obiectelor intre doua apeluri ale programului. Obiectele sunt in mod normal stocate in RAM si se pierd cand aplicatia se inchide (sau mai exact cand memoria este eliberata de catre garbage collector); folosind clase specializate putem salva obiectele cu posibilitatea incarcarii lor ulterioare. Termenul de *lightweight* provine de la faptul ca salvarea/incarcarea obiectelor nu se face automat de catre JVM in urma marcarii obiectului de catre programator, ci acesta din urma este cel care trebuie sa salveze/incarce obiectele dorite in mod explicit prin crearea unui stream specializat si folosirea metodelor aferente.

Pentru ca un obiect sa poate fi salvat el trebuie sa implementeze interfata **Serializable**. Aceasta nu contine nici campuri nici metode, ci marcheaza obiectul ca fiind salvabil (**obiectele care nu sunt instanceof Serializable nu se pot salva!**). Salvarea presupune scrierea starii obiectului intr-un stream, care poate avea in spate, de exemplu, un fisier (si atunci obiectul va fi scris pe HDD) sau o conexiune de retea (si atunci obiectul va fi transferat altui masini virtuale).

A serializa un obiect inseamna a-i transforma starea curenta intr-o succesiune de octeti, care sa permita refacerea ulterioara a acelei stari. Masina virtuala automatizeaza procesul de salvare a starii obiectului prin urmarirea in mod recursiv a referintelor catre alte obiecte si salvarea acestora la randul lor. Spre exemplu, daca un obiect *GrupPersoane* contine un tablou de obiecte *Persoana*, toate obiectele *Persoana* vor fi serializate la randul lor. **Atentie insa! Toate clasele care sunt implicate in acest mecanism recursiv trebuie sa implementeze Serializable!** (in exemplul anterior, atat clasa *GrupPersoane* cat si clasa *Persoana*).

**Nota:** serializarea nu este legata exclusiv de fisiere, ci este folosita si pentru transmiterea obiectelor prin retea, catre o alta aplicatie (masina virtuala) care le reconstituie starea.

Pentru salvarea/incarcarea unui obiect, Java dispune de clasele **ObjectOutputStream** si **ObjectInputStream**. Constructorul lui *ObjectOutputStream* primeste ca parametru orice obiect de tip *OutputStream* (sau subclasa), asadar pentru a salva un obiect pe HDD vom construi un *FileInputStream* pe care il vom pasa ca parametru constructorului:

```
FileOutputStream fos=new FileOutputStream(new File("c:\\java\\file1"));
 ObjectOutputStream oos=new ObjectOutputStream(fos);
 oos.writeObject(obiectulDoritCareImplementeazaSerializable);
 oos.close();
```

In urma acestei operatiuni:

- se scriu in stream:
  - numele clasei obiectului
  - versiunea clasei (astfel incat sa se detecteze situatiile in care se incarca obiecte ale unei clase care intre timp a fost modificata)
  - valorile tuturor campurilor de instanta ale obiectului, fie ele primitive sau tablouri
  - daca exista referinte catre alte obiecte, starea obiectelor referite va fi si ea salvata, in mod recursiv
- NU se scriu in stream
  - definitia clasei - masina virtuala care incarca starea obiectului trebuie sa dispuna de definitia clasei in cauza, incarcand fisierul *.class* corespunzator prin mecanismele deja discutate cu alta ocazie
  - campurile statice - ele apar in clasei si nu instantelor
  - campurile marcate ca **transient** (calificator care permite programatorului sa exclude explicit un camp din forma serializata a unui obiect)

Pe langa obiecte, *ObjectInputStream* si *ObjectOutputStream* pot lucra si cu primitive, prin intermediul unor metode ca **readLong()**, **readChar()**, **writeFloat()** etc.

Incarcarea unui obiect scris anterior se poate realiza definind sirul de obiecte stream de pe ramura de Input si apeland metoda *readObject()* din clasa *ObjectInputStream*; aceasta returneaza un *Object*, si de aceea va fi necesara in general o conversie explicita la tipul de date dorit:

```
ObjectInputStream ois=new ObjectInputStream(
    new FileInputStream(new File("c:\\java\\\\obiect_salvat")));
Date d=(Date)ois.readObject();
ois.close();
```

Deserializarea (incarcarea) unui obiect se face analog cu instantierea, insa de aceasta data nu se apeleaza niciun constructor: este alocata memorie, sunt ignorate campurile statice si cele *transient*, iar restul de memorie este initializata cu datele citite din stream.

Iata un exemplu complet, care creaza un obiect de tip data, il salveaza intr-un fisier de pe HDD (creat anterior) si apoi il incarca:

```
File f=new File("c:\\\\obiect_salvat");
f.createNewFile();
Date d=new Date(),dd;
ObjectOutputStream oos=new ObjectOutputStream(new FileOutputStream(f));
System.out.println("S-a salvat obiectul "+d);
oos.writeObject(d);
ObjectInputStream ois=new ObjectInputStream(new FileInputStream(f));
dd=(Date)ois.readObject();
System.out.println("S-a incarcat obiectul "+dd);
```

## 8.4. BIBLIOGRAFIE

- Exceptii: <http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>
- Lucrul cu sistemul de fisiere: <http://docs.oracle.com/javase/tutorial/essential/io/index.html>

## 8.5. Anexa 1 - corespondente java.io - java.nio.file (cf. Java tutorial)

java.io.File functionality	java.nio.file functionality
java.io.File	java.nio.file.Path
java.io.RandomAccessFile	The SeekableByteChannel functionality.
File.canRead, canWrite, canExecute	Files.isReadable, Files.isWritable, and Files.isExecutable. On UNIX file systems, the <a href="#">Managing Metadata (File and File Store Attributes)</a> package is used to check the nine file permissions.
File.isDirectory(), File.isFile(), and File.length()	Files.isDirectory(Path, LinkOption...), Files.isRegularFile(Path, LinkOption...), and Files.size(Path)
File.lastModified() and File.setLastModified(long)	Files.getLastModifiedTime(Path, LinkOption...) and Files.setLastModifiedTime(Path, FileTime)
The File methods that set various attributes: setExecutable, setReadable, setReadOnly, setWritable	These methods are replaced by the Files method setAttribute(Path, String, Object, LinkOption...).
new File(parent, "newfile")	parent.resolve("newfile")
File.renameTo	Files.move
File.delete	Files.delete
File.createNewFile	Files.createFile
File.deleteOnExit	Replaced by the DELETE_ON_CLOSE option specified in the createFile method.
File.createTempFile	Files.createTempFile(Path, String, FileAttributes<?>), Files.createTempFile(Path, String, String, FileAttributes<?>)
File.exists	Files.exists and Files.notExists
File.compareTo and equals	Path.compareTo and equals
File.getAbsolutePath and getAbsoluteFile	Path.toAbsolutePath

Studentul poate utiliza prezentul material si informatiile continute in el exclusiv in scopul asimilarii cunostintelor pe care le include, fara a afecta dreptul de proprietate intelectuala detinut de autor.

File.getCanonicalPath and getCanonicalFile	Path.toRealPath or normalize
File.toURI	Path.toURI
File.isHidden	Files.isHidden
File.list and listFiles	Path.newDirectoryStream
File.mkdir and mkdirs	Path.createDirectory
File.listRoots	FileSystem.getRootDirectories
File.getTotalSpace, File.getFreeSpace, File.getUsableSpace	FileStore.getTotalSpace, FileStore.getUnallocatedSpace, FileStore.getUsableSpace, FileStore.getTotalSpace

## 8.6. Anexa 2 (informativa): Preferences API

Daca dorim ca anumite setari ale aplicatiei sa se pastreze intre doua rulari ale acesteia, este necesara memorarea informatiilor de configurare intr-un spatiu de stocare persistent. Daca am alege ca solutie un fisier, am fi nevoiti fie sa includem in aplicatie (hard-coded) calea catre el, fie sa memoram intr-un alt loc aceasta cale (ceea ce duce la o problema recursiva). Rezolvarea acestui gen de dilema este dependenta de sistemul de operare, insa orice sistem de operare pune la dispozitia aplicatiilor o modalitate de a-si stoca persistent informatiile de configurare (ex: Windows registry).

Prin intermediul clasei Preferences, programatorul are la dispozitie o modalitate simpla si independenta de sistemul de operare pentru stocarea persistenta a informatiilor de configurare ale aplicatiei. API-ul pus la dispozitie de catre aceasta clasa permite definirea de setari de sistem sau per-user, modalitatea efectiva de memorare a lor depinzand de sistemul de operare gazda, insa fara ca acest aspect sa preocupe programatorul. De exemplu, in Windows, preferintele utilizatorului se vor gasi in registry, in HKEY\_CURRENT\_USER/Software/JavaSoft/Prefs/numele\_pachetului\_clasei, iar cele de sistem in HKEY\_LOCAL\_MACHINE/SOFTWARE/JavaSoft/Prefs.

Preferintele sunt stocate arborescent, incepand cu user root/system root (radacina ierarhiei de preferinte). Numele complet al unui nod este format din succesiunea numelor de noduri de la radacina pana la el, separate prin /, la fel ca in cazul sistemului de fisiere (ex: /app1/windowSettings). Fiecare nod are asociata o lista de perechi cheie → valoare, unde cheia este de tip String iar valoarea poate fi String, boolean, int, long, float, double, tablou de byte.

Metode statice:

- **Preferences.userNodeForPackage(Class c)** – returneaza un obiect Preferences pentru userul curent si pachetul clasei specificate
- **Preferences.systemNodeForPackage(Class c)** – returneaza un obiect Preferences pentru setarile de sistem si pachetul clasei specificate
- **Preferences.userRoot()** – returneaza obiectul de tip Preferences corespunzator radacinii ierarhiei de preferinte pentru userul curent
- **Preferences.systemRoot()** – returneaza obiectul de tip Preferences corespunzator radacinii ierarhiei de preferinte de sistem

Odata obtinut un obiect de tip Preferences, i se pot apela metode de instanta, precum:

- **put(String cheie, String valoare)** – memoreaza o asociere intre cheia si valoarea specificata
- **get(String cheie, String default)** – citeste valoarea cheii specificate din nodul curent, iar daca citirea nu se poate efectua (din motive de indisponibilitate a back-end-ului de stocare) este returnata valoarea default specificata
- **putBoolean(String cheie, boolean valoare), putInt(String cheie, int valoare), putDouble, putFloat, putLong, putByteArray** – memoreaza valoarea ceruta in corespondenta cu cheia specificata
- **getBoolean(String cheie, String default), getInt(String cheie, int default), getDouble, getFloat, getLong, getByteArray** – analog cu get(String, String)
- **childrenNames()** – returneaza un array de String cu numele nodurilor fiu ale nodului curent
- **name()** – returneaza numele nodului
- **isUserNode()** – verifica daca nodul contine preferinte per-user sau de sistem

- **keys()** - returneaza un array de String cu cheile care au asociere in nodul curent
- **clear()** – sterge toate asocierile cheie-valoare ale nodului curent
- **exportNode(OutputStream o), exportSubtree(OutputStream o), importPreferences(InputStream i)** – permite exportarea/importarea preferintelor in format XML

Exemplu: dorim ca o aplicatie sa-si salveze dimensiunile ferestrei principale la iesire si sa le incarce la pornire.

```
// in constructorul JFrame-ului:  
public GUI(){  
    [...]  
    Preferences pref = Preferences.userNodeForPackage(getClass());  
    setSize(pref.getInt("width",100),pref.getInt("height",100));  
}  
// ...iar in metoda windowClosing() - tratarea evenimentului de inchidere a ferestrei:  
public void windowClosing(WindowEvent we){  
    Preferences pref = Preferences.userNodeForPackage(getClass());  
    pref.putInt("width",getWidth());  
    pref.putInt("height",getHeight());  
    System.exit(0);  
}
```