Capitolul 4

Clase si metode abstracte

Sa reluam exemplul din capitolul anterior in care derivam din clasa Forma2D alte doua clase: Triunghi si Dreptunghi. Sa presupunem ca vrem sa derivam un numar mai mare de clase: Dreptunghi, Triunghi, Elipsa, Trapez si asa mai departe. Putem observa ca toate aceste clase au doua elemente comune si anume *Suprafata* si *Perimetrul* sau *Circumferinta*. Pentru a putea lucra usor cu un sir de obiecte care pot fi sau Triunghi sau Elipsa, este util ca acestea sa faca parte din clase ce au o superclasa si anume Forma2D. In aceste conditii, ar trebui ca *Forma2D* sa implementeze functiile *Suprafata* si *Perimetrul*, insa intrebarea este cum daca forma nu este una anume, deci nu stim exact cum vor fi implementate aceste doua metode.

In aceste situatii Java permite utilizarea metodelor *abstracte*.

O metoda abstracta nu are un corp definit, ci doar o semnatura, iar declaratia functiei se incheie cu „;”. Mai jos sunt o serie de reguli pentru a defini metode abstracte si clase abstracte:

Orice clasa ce contine o metoda abstracta este in mod automat abstracta, asa ca trebuie declarata ca atare.

O clasa *abstracta* nu poate fi instantiata.

O subclasa a unei clase abstracte poate fi instantiata doar daca suprascrie metodele abstracte ale parintelui si ofera o implementare pentru toate aceste metode. Aceste clase copil se numesc si *concrete* pentru a sublinia ca nu sunt abstracte.

Daca o clasa copil a unei superclase abstracte nu implementeaza toate metodele abstracte pe care le mosteneste, atunci si clasa copil este abstracta si va fi declarata ca atare.

Metodele declarate *static*, *private* sau *final* nu pot fi abstracte deoarece niciuna din acestea nu poate fi suprascrisa de subclase. De asemenea o clasa declarata *final* nu poate contine metode *abstracte*.

O clasa poate fi declarata *abstract* chiar daca nu contine metode abstracte. De obicei aceasta semnifica faptul ca o clasa este incompleta si este clasa parinte pentru una sau mai multe clase copil.

Mai jos avem une exemplu de folosire a claselor abstracte

```
public abstract class Forma
{
  public String categorie;
  public abstract double Suprafata();
  public abstract double Perimetrul();   //in loc de corp avem ";"
}

class Cerc extends Forma
{
      public static final double PI = 3.1415926535897;
      protected double raza;
      public Cerc(double r)
      {
```

```
            this.raza = r;
            categorie = "Cerc";
      }
      public double getRaza()
      {
            return raza;
      }
      //metodele abstracte care "prind forma":

      public double Suprafata()
      {
            return PI*raza*raza;
      }
      public double Perimetrul()
      {
            return 2*PI*raza;
      }
}

class Dreptunghi extends Forma
{
      protected double latime, inaltime;
      public Dreptunghi(double latime, double inaltime)
      {
            categorie = "Dreptunghi";
            this.latime = latime;
            this.inaltime = inaltime;
        }
      public double getLatime()
      {
            return latime;
      }
      public double getInaltime()
      {
            return inaltime;
      }
      //metodele abstracte care "prind forma":

      public double Suprafata()
      {
            return latime*inaltime;
      }
      public double Perimetrul()
      {
            return 2*(latime + inaltime);
      }
}
```

```
class AbstractDemo
{
      public static void main(String args[])
      {
            //declaram un sir de clase abstracte
            Forma[] sir_forme = new Forma[4];
            //instantiam diverse obiecte din sir
            sir_forme[0] = new Cerc(2.0);
            sir_forme[1] = new Dreptunghi(1.0, 3.0);
            sir_forme[2] = new Dreptunghi(4.0, 2.0);
            sir_forme[3] = new Cerc(5.0);

            double suprafata_totala = 0;
            for(int i = 0; i < sir_forme.length; i++)
            {
                suprafata_totala += sir_forme[i].Suprafata();
                //putem afisa aceste date deoarece sunt din clasa
                //parinte

                System.out.println("Categoria: " +
                  sir_forme[i].categorie);
                System.out.println("Perimetrul: " +
                  sir_forme[i].Perimetrul());
                System.out.println("Suprafata: " +
                  sir_forme[i].Suprafata());

                //nu putem accesa aceste date, deoarece sunt specifice
                //doar unei subclase si nu parintelui
                //System.out.println("PI este: " + sir_forme[i].PI);

            }
            System.out.println("Suprafata totala: " +
            suprafata_totala);
      }
}
```

In urma rularii acestui program rezultatul este:

```
Categoria: Cerc
Perimetrul: 12.5663706143588
Suprafata: 12.5663706143588
Categoria: Dreptunghi
Perimetrul: 8.0
Suprafata: 3.0
Categoria: Dreptunghi
Perimetrul: 12.0
Suprafata: 8.0
Categoria: Cerc
Perimetrul: 31.415926535897
```

```
Suprafata: 78.5398163397425
Suprafata totala: 102.1061869541013
```

Clasa Forma este declarata abstract si ca atare nu va pute fi instantiata. Totusi obiectele, sau sirul de obiecte este de tip Forma, pentru ca instantierea se petrece ulterior cu o subclasa a parintelui si anume Cerc:

```
sir_forme[0] = new Cerc(2.0);
```

Clasele Cerc cat si Dreptunghi nu sunt abstracte deoarece ambele implementeaza toate metodele declarate abstracte ale clasei Forma.

Cele doua metode *Suprafata* si *Perimetrul* sunt declarate cu un cuvant cheie *abstract* in clasa *Forma* si nu au corp, deci se vor termina cu „;" ca orice declaratie obisnuita.

Pe de alta parte cele doua clase copil Cerc si Dreptunghi, implementeaza ambele metode:

```
public double Suprafata()
      {
            return PI*raza*raza;
      }
      public double Perimetrul()
      {
            return 2*PI*raza;
      }
```

Pe langa aceste doua metode ambele clase pot contine propriile metode, separat una de cealalta. De exemplu Cerc mai implementeaza urmatoarea metoda:

```
      public double getRaza()
      {
            return raza;
      }
```

Atunci cand folosim obiectele acestor clase trebuie sa avem grija, deoarece, desi toate obiectele din sirul `sir_forme` sunt de tipul Forma, primul si ultimul obiect sunt de tipul Cerc, iar al doilea si al treilea sunt de tipul Dreptunghi.

Daca un obiect este de tip Forma, nu putem apela un membru (desi public) al clasei Cerc. O astfel de instructiune este gresita

```
System.out.println("PI este: " + sir_forme[i].PI);
```

si produce urmatoarea eroare de compilare:

```
Forma.java:86: cannot find symbol
symbol  : variable PI
location: class Forma
                  System.out.println("PI este: " + sir_forme[i].PI);
```

Mai mult, in clasa parinte Forma, pot exista pe langa metode abstracte si alte metode concrete ca de exemplu aceasta:

```
  public void ShowTipForma()
       {
            System.out.println("Forma este de tip: " + categorie);
       }
```

Exemplu de folosire a acestei functii:

```
for(int i = 0; i < sir_forme.length; i++)
```

```
{
    sir_forme[i].ShowTipForma();
```

Interfete

In OOP este indicat sa definim ce trebuie sa faca o functie fara a da toate detaliile despre cum trebuie sa faca acele sarcini. O metoda abstracta este raspunsul la aceasta problema. In unele cazuri in care clasa nu are nevoie de metode concrete, se pot crea interfete, ce separa total implementarea unei clase de definitia metodelor acesteia.

Interfetele sunt asemanatoare cu clasele abstracte, dar intr-o interfata, nici o metoda nu are voie sa aiba corp, adica implementare. Mai mult, o clasa poate *implementa* una sau mai multe interfete.

Pentru ca implementarea unei interfete sa fie corecta, o clasa trebuie sa ofere implementari acelor metode declarate in interfata. Evident, fiecare clasa poate avea propria implementare pentru metodele interfetei. Mai jos este modul general de a declara o interfata:

```
acces interface nume_interfata
 {
      tip_data nume_metoda1(param-list);
      tip_data nume_metoda2(param-list);
      tip_data var1 = valoare;
      tip-_data var2 = valoare;
      // ...
      tip_data nume_metodaN(param-list);
      tip_data varN = valoare;
}
```

unde *acces* este *public* sau nu este folosit deloc. Metodele sunt declarate folosind doar semnatura lor exact ca niste metode abstracte. Implicit metodele sunt *public*. Variabilele declarate intr-o interfata nu sunt variabile de instanta, ci trebuie declarate *public*, *final* si *static* si trebuie initializate.

In exemplul de mai sus, clasa Forma poate implementa o interfata care poate specifica centrul unei forme geometrice:

```
public interface Centrat
{
    void setCentru(double x, double y);
    double getCentruX();
    double getCentruY();
}
```

De asemenea o interfata nu poate fi instantiata si nu poate defini un constructor.

Mostenirea unor interfete

Interfetele pot mosteni alte interfete, asemanator claselor, folosind cuvantul cheie **extends**. Cand o interfata mosteneste o alta interfata, va mosteni toate metodele abstracte ale interfetei parinte, putand defini si metode abstracte noi si alte constante. Spre deosebire de clase, o interfata poate mosteni mai multe interfete:

```
public interface Positionabil extends Centrat
```

```
{
    void setColDreaptaSus(double x, double y);
    double getDreaptaSusX();
    double getDreaptaSusY();
}
public interface Transformabil extends Scalabil, Translatabil,
Rotatabil {}
public interface Forma extends Positionabil, Transformabil {}
```

Implementarea unei interfete

Asa cum o clasa foloseste **extends**, pentru a mosteni o clasa parinte, se poate folosi **implements** pentru a mosteni interfete.

Atunci cand o clasa mosteneste o interfata, ofera o implementare tuturor metodelor interfetei. Daca o clasa implementeaza o interfata fara a oferi o implementare pentru fiecare metoda, va mosteni acele metode abstracte, si devine la randul sau abstracta.

Forma generala a unei implementari este:

```
acces class nume_clasa extends superclasa implements interfata {
// corpul clasei
}
```

Acces este fie *public* fie nu este declarat. Clauza *extends* este optionala, in cazul in care clasa originala extinde alte clase „obisnuite”. Ce intereseaza in declaratia de mai sus este clauza *implements* ce permite implementarea unei interfete. Mai jos se gaseste un exemplu de utilizare al interfetelor:

```
abstract class Forma
{
  public String categorie;
  public abstract double Suprafata();
  public abstract double Perimetrul();   //in loc de corp avem ";"
  public void ShowTipForma()
       {
            System.out.println("Forma este de tip: " + categorie);
       }
}
class Dreptunghi extends Forma
{
      protected double latime, inaltime;
      public Dreptunghi(double latime, double inaltime)
      {
            categorie = "Dreptunghi";
            this.latime = latime;
            this.inaltime = inaltime;
        }
      public double getLatime()
      {
```

```
            return latime;
        }
        public double getInaltime()
        {
            return inaltime;
        }
        //metodele abstracte care "prind forma":

        public double Suprafata()
        {
            return latime*inaltime;
        }
        public double Perimetrul()
        {
            return 2*(latime + inaltime);
        }
    }

    public interface Centrat
    {
        void setCentru(double x, double y);
        double getCentruX();
        double getCentruY();
    }

    class Dreptunghi_Centrat extends Dreptunghi implements Centrat
    {
        //campuri ale clasei Dreptunghi_Centrat
        private double cx, cy;

        //constructorul
        public Dreptunghi_Centrat (double cx, double cy, double latime,
double inaltime)
        {
            super(latime, inaltime);
            this.cx = cx;
            this.cy = cy;
        }

        //Daca implementam interfata Centrat
        //trebuie sa ii implementam metodele
        public void setCentru(double x, double y)
        {
            cx = x;
            cy = y;
        }
        public double getCentruX()
        {
```

```
            return cx;
        }
        public double getCentruY()
        {
            return cy;
        }
    }
    class DemoInterfete
    {
        public static void main(String args[])
        {
            Dreptunghi_Centrat drept_centrat = new
            Dreptunghi_Centrat(2.4,3.5,6,9);
            Centrat c = (Centrat) drept_centrat;
            double cx = c.getCentruX(); //coordonatele centrului
            double cy = c.getCentruY();
            //calculam distanta de la origine
            double dist = Math.sqrt(cx*cx + cy*cy);
            System.out.println("Suprafata " +
            drept_centrat.Suprafata());
            System.out.println("Distanta de calibrat : " + dist);
        }
    }
```

Dupa cum se observa, interfata declarata este `Centrat`, iar clasa care o implementeaza este `Dreptunghi_Centrat`. Deoarece vorbim despre o figura geometrica de tip dreptunghi aceasta clasa, `Dreptunghi_Centrat` va mosteni de asemenea si clasa `Dreptunghi`.

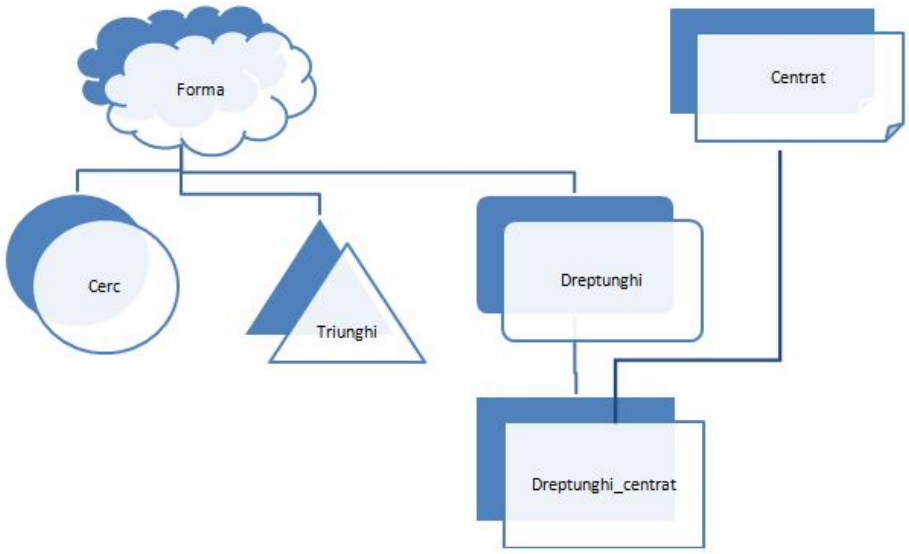


Figura 4.1 Diagrama claselor din exemplul pentru interferte

Pentru a intelege mai bine toata ierarhia de clase care sunt folosite, avem diagrama din figura 4.1.

In aceasta figura se poate vedea cum interfata este implementata in Dreptunghi_centrat, iar clasa abstracta Forma este mostenita de Dreptunghi. Implicit, atunci si Dreptunghi_centrat va mosteni proprietatile clasei Forma.

In continuare vom prezenta o dezbatere pe seama diferentelor si asemanarilor dintre interfete si clasele abstracte.

Interfete si clase abstracte

Atunci cand definim tipuri de data abstracte (de exemplu Forma) care pot avea subclase (de exemplu Cerc, Dreptunghi etc) vom avea de ales daca acel tip de data este o clasa abstracta sau o interfata.

O interfata este recomandata pentru ca orice clasa poate sa o implementeze, chiar si atunci cand acea interfata extinde o alta superclasa care nu are nici o legatura cu interfata.

Totusi, cand o interfata contine multe metode poate deveni greoi implementarea tuturor metodelor, in continuu pentru fiecare clasa ce o implementeaza.

Din acest punct de vedere putem spune ca o clasa abstracta nu trebuie sa fie in intregime abstracta, ea poate contine o implementare cel putin partiala, iar subclasele o pot folosi pe aceasta. Pe de alta parte, o clasa care extinde o clasa abstracta nu mai poate extinde (mosteni) o alta clasa, ceea ce duce la unele dificulati.

O alta diferenta dintre clase abstracte si interfete este legata de compatibilitate. Daca definim interfata si o adaugam intr-o librarie, iar mai apoi adaugam o metoda acelei interfete, atunci clasele care implementau versiunea anterioara de interfata vor fi stricate (nu implementeaza corect interfata noua). Se spune ca am stricat compatibilitatea. Daca folosim clase abstracte pe de alta parte, putem adauga metode nonabstracte fara ca si clasele ce o mostentesc sa fie alterate de aceasta.

Mai multe exemple cu interfete

Folosirea referintelor la o interfata

Desi nu putem instantia o interfata, se poate lucra cu un obiect care face referinta la o interfata. Atunci cand apelam o metoda a unui obiect prin referinta la interfata, este de fapt vorba de metoda implementata de obiect. Pentru a intelege mai bine conceptul avem interfata de mai jos:

```
public interface Serii
{
      int getNext(); //numarul urmator din serie
      void reset(); // resetez
      void setStart(int x); // cu ce incep
}

class DinDoiinDoi implements Serii
{
```

```
	int start;
	int val;
	DinDoiinDoi()
	{
		start = 0;
		val = 0;
	}
	public int getNext()
	{
		val += 2;
		return val;
	}
	public void reset()
	{
		start = 0;
		val = 0;
	}
	public void setStart(int x)
	{
		start = x;
		val = x;
	}
}

class DinTreiinTrei implements Serii
{
	int start;
	int val;
	DinTreiinTrei()
	{
		start = 0;
		val = 0;
	}
	public int getNext()
	{
		val += 3;
		return val;
	}
	public void reset()
	{
		start = 0;
		val = 0;
	}
	public void setStart(int x)
	{
		start = x;
		val = x;
	}
```

```
}

class DemoSerii
{
	public static void main(String args[])
	{
		DinDoiinDoi doiobj = new DinDoiinDoi ();
		DinTreiinTrei treiobj = new DinTreiinTrei ();
		Serii ob;
		for(int i=0; i < 10; i++)
		{
			ob = doiobj;
			System.out.println("Urmatorul numar par " +
			ob.getNext()); //apelez metoda definita de interfata
					//implementata in clasa DinDoiinDoi
			ob = treiobj;
			System.out.println("Urmatorul numar ternar " +
			ob.getNext()); //apelez metoda definita de interfata
					//implementata in clasa DinTreiinTrei
		}
	}
}
```

Cele doua clase `DinDoiinDoi` si `DinTreiinTrei` implementeaza aceeasi interfata si anume `Serii`. Aceasta va fi referita prin obiectul *ob,* insa acest obiect nu poate fi instantiat cu o interfata. Totusi, el va „prinde forma" atunci cand este referentiat prin `doiobj`, fapt perfect legal pentru ca putem spune ca „sunt de acelasi tip".La fel putem avea instantierea lui ob cu un obiect de tipul `DinTreiinTrei` pe acelasi principiu: `ob = treiobj;`. Observam ca interfata nu contine deocamdata variabile, insa ambele clase folosesc aceleasi tipuri de variabile si anume `int start;` si `int val;` pentru a contoriza urmatorul numar si pozitia de inceput. Oare aceste variabile pot fi declarate in interfata?

Variabile in interfete

Variabilele in interfete trebuie sa fie *static*, *public* si *final*. Practic este vorba de constante.

```
interface IConst
{
	int MIN = 0;
	int MAX;
	String ERRORMSG = "Eroare de limita";
}
class DemoIConst implements IConst
{
	public static void main(String args[])
	{
		int nums[] = new int[MAX];
```

```
            for(int i=MIN; i < 11; i++)
            {
                  if(i >= MAX) System.out.println(ERRORMSG);
                  else
                  {
                        nums[i] = i;
                        System.out.print(nums[i] + " ");
                  }
            }
      }
}
```

Orice incercare de modificare a valorii unei constante va esua la compilare

```
var.java:12: cannot assign a value to final variable MAX
                MAX=20;
                ^
1 error
```

De asemena, orice nerespectare a conditiilor de mai sus va aduce cu sine aparitia diverselor erori de compilare corespunzatoare.

Exceptii

O exceptie este o eroare care apare la un moment dat in timpul rularii programului. Folosind mecanismul de tratare al erorilor oferit de Java, putem sa controlam logic ce se intampla atunci cand apare o eroare. Un avantaj pe care acest mecanism il ofera este ca automatizeaza codul pentru tratarea erorilor, care initial trebuia introdus „manual” de programator, adica prevazand fiecare situatie in parte.

De exemplu, atunci cand o metoda esueaza, este returnat un cod de eroare, iar corespunzator acelui cod, se verifica ce anume nu a mers sau a esuat. Acest mod de rezolvare este greoi si necesita atentie din partea programatorului. Mecanismul oferit de Java este ca o plasa de protectie, care nu permite ca programatorul sa greseasca, deoarece cand apare o eroare, automat o functie *handler* de exceptii este apelata, fara sa avem grija codului returnat de functii.

Un alt motiv pentru care tratarea exceptiilor este importanta, este ca Java defineste un standard pentru diversele tipuri de exceptii de exemplu fisierul ce nu exista sau diviziune cu zero.

Cum functioneaza?

Atunci cand o eroare apare intr-o metoda, metoda creaza un obiect si il preda sistemului runtime Java. Obiectul poarta denumirea de obiectul exceptiei sau mai simplu, exceptie. El contine informatie despre eroare, inclusiv tipul erorii si starea programului atunci cand a aparut eroarea. Creearea unui obiect exceptie si predarea acestuia catre sistem se numeste aruncarea unei exceptii.

Dupa ce o metoda arunca o exceptie, sistemul runtime incearca sa gaseasca prima functie sau bloc care trateaza acesta eroare. Multimea de aceste posibile functii sau blocuri care sa trateze eroarea ordonata intr-o lista a evenimentelor se numeste *call stack* sau stiva de apel.

Sistemul cauta in stiva de apel o metoda care contine un bloc ce trateaza exceptia. Acest bloc se numeste *handler* de exceptie. Cautarea incepe cu metoda in care apare eroarea si continua in ordine inversa a apelurilor cu toata stiva de apel.

Se spune ca handler-ul de exceptie „prinde” exceptia. Daca sistemul runtime a epuizat stiva de apel fara a gasi un handler de exceptie potrivit, executia se incheie.

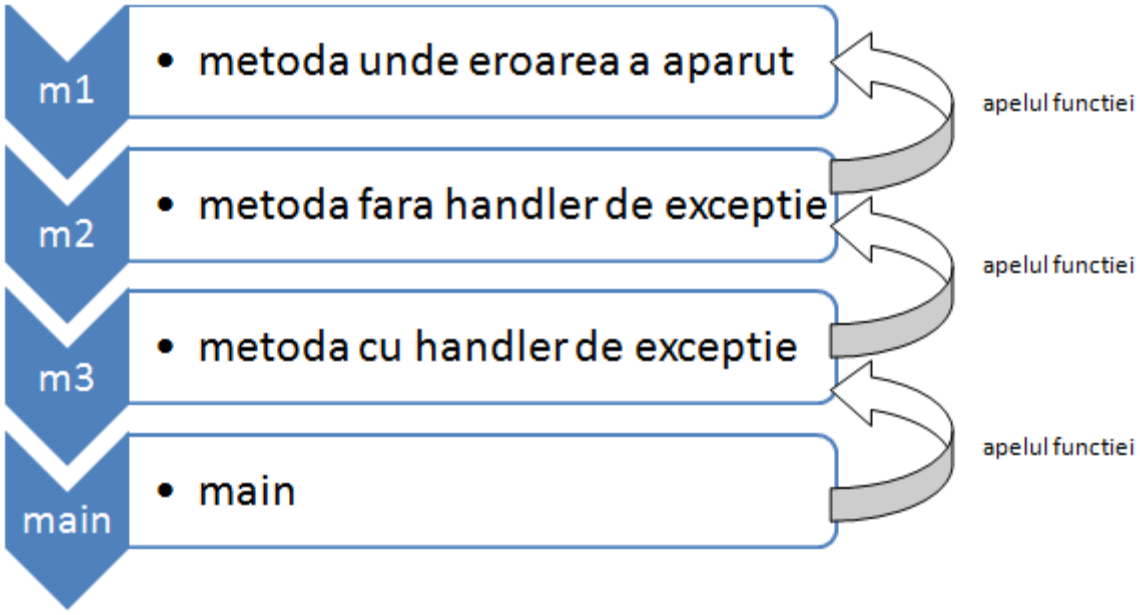


Figura 4.2 Stiva de apel si incercarea de tratare a erorii

Sa presupunem ca apare eroarea in cadrul functiei *m1*. Metoda va fi aruncata in functia care a apelat metoda *m1* si anume *m2*. Problema este ca functia *m2* nu contine nici un bloc de prindere a erorii astfel eroarea este aruncata mai departe in functia care a apelat *m2* si anume *m3*. Acum exista doua posibilitati, *m3* sa prinda eroarea si atat sau sa o arunce mai departe. Vom detalia in continuare.

Se recomanda ca o metoda sa prinda sau sa specifice faptul ca poate genera o anumita exceptie, daca este cazul. Anume, prinderea erorii se specifica prin cuvantul cheie *catch*, iar faptul ca arunca o exceptie se specifica prin cuvantul cheie *throws*.

Folosirea handlerului de exceptii try catch

Forma generala a acestui mecanism este:

```
try
{
      // blocul in care eroarea poate apare
}
catch (ExcepType1 exOb)
{
      // handler pentru ExcepType1
}
catch (ExcepType2 exOb)
{
      // handler pentru ExcepType2
}
```

In aceasta declaratie, in blocul specificat de *try* va sta codul obisnuit si anume instructiunile pe care dorim sa le implementam, iar in *catch* care si are forma unei functii cu un parametru va sta codul de tratare al exceptiilor. Se arata in aceasta declaratie ca pot fi mai multe handler-e asociate unui try. Aceasta inseamna ca mai multe tipuri de erori pot sa apara intr-un bloc try iar ele sunt tratate separat. Ne putem gandi ca in functie de tipul erorii care apare, aceste handler-e sunt suprascrise, iar apelarea lor se face in functie de tipul exceptiei.

Sa analizam un exemplu simplu de tratare a exceptiilor

```
class ExceptionDemo1
{
	public static void main(String args[])
	{
		int nums[] = new int[4];
		try
		{
			System.out.println("Inainte de generarea erorii.");
			// Generam o exceptie prin depasirea limitei sirului
			nums[5] = 7;
			System.out.println("Eroarea a aparut deja.
			Instructiunea   aceasta nu va mai fi executata.");
		}
		catch (ArrayIndexOutOfBoundsException exc)
		{
			// prindem si tratam exceptia
			System.out.println("Depasirea limitei sirului!");
		}
		System.out.println("Dupa blocul try catch.");
	}
}
```

Codul care monitorizeaza erorile va sta in blocul *try*. Atunci cand apare eroarea, adica atunci cand se incearca executia `nums[5] = 7;` exceptia este este aruncata din blocul try si se iese din acest bloc. De aceea, instructiunea ce ar fi urmat nu se mai executa, ci se va executa blocul catch aferent. Mai precis, se executa instructiunea `System.out.println("Depasirea limitei sirului!");`. Dupa ce se termina de executat blocul catch se trece la urmatoarea instructiune si anume `System.out.println("Dupa blocul try catch.");`.

Mai departe putem exemplifica pentru modelul din figura 4.2 cum exceptiile pot fi prinse si eventual, tratate in blocul catch scris in alta functie diferita de cea care genereaza eroarea:

```
class ExceptionDemo1
{
	public static void genException()
	{
		int nums[] = new int[4];
			System.out.println("Inainte de generarea erorii.");
			// Generam o exceptie prin depasirea limitei sirului
			nums[7] = 10;
```

```
                        System.out.println("Eroarea a aparut deja.
Instructiunea   aceasta nu va mai fi executata?");

            }
      }

      class ExceptionDemo2
      {
            public static void main(String args[])
            {
                  try
                  {
                        ExceptionDemo1.genException();
                  }
                  catch (ArrayIndexOutOfBoundsException exc)
                  {
                        // prindem si tratam exceptia
                        System.out.println("Depasirea limitei sirului!");
                  }
                  System.out.println("Dupa blocul try catch.");
            }
      }
```

Acest program va produce acelasi afisaj ca cel de sus, insa, din cauze diferite. De data aceasta, exceptia este generata in clasa `ExceptionDemo1m` mai exact, in functia `genException`. Exceptia nu mai este tratata in cadrul functiei ci este aruncata „mai departe” in functia care efectueaza apelul, si anume, *main* din clasa `ExceptionDemo2`.

In cazul in care o exceptie nu este prinsa de nici una din functiile din call stack, atunci exceptia este prinsa de JVM si executia programului se incheie brusc.

Acest lucru nu este prea placut pentru utilizatorii programului scris fara un mecanism de tratare a exceptiilor si ii va deceptiona.

Mai sus am mentionat ca exceptiile pot avea mai multe tipuri. Daca blocul de catch este scris pentru o alta exceptie decat cea care va apare probabil in blocul try, atunci tot scopul mecanismului de tratare a exceptiilor dispare.

```
      class ExceptionTypeMismatch
      {
            public static void main(String args[])
            {
                  int nums[] = new int[4];
                  try
                  {
                        System.out.println("Inainte ca exceptia sa apara.");
                        // generam o exceptie
                        nums[6] = 4;
                        System.out.println("nu va ajunge aici.");
                  }
                  //nu aceasta este tipul de eroare care va apare
                  catch (ArithmeticException exc)
```

```
            {
                  //prind exceptia
                  System.out.println("In afara limitelor!");
            }
            System.out.println("Dupa try catch.");
      }
}
```

In exemplul de mai sus se incearca tratarea exceptiei de tipul `ArithmeticException` si in blocul *try* apare alt tip de exceptie. Ceea ce se intampla la rulare, este usor de prevazut, si anume, faptul ca exceptia *ArrayIndexOutOfBoundException* ce apare va duce la oprirea programului. Este ca si cand eroarea nu este tratata.

In cazul acesta rezolvarea rapida ar fi sa tratam ambele tipuri de exceptii. Alta rezolvare implica modul in care sunt implementate exceptiile in Java, pe care il vom analiza mai tarziu.

```
class ExcMultipleDemo
{
      public static void main(String args[])
      {

            int numar[] = { 5, 3, 12, 1, 20, 27, 10, 56 };
            int deimp[] = { 1, 0, 3, 4, 0, 5 };
            for(int i=0; i<numar.length; i++)
            {
                  try
                  {
                        System.out.println(numar[i] + " / " +
                        deimp[i] + " este " + numar[i]/deimp[i]);
                  }
                  catch (ArithmeticException aex)
                  {
                        System.out.println("impart la zero la pasul "+
                        i+"!");
                  }
                  catch (ArrayIndexOutOfBoundsException iex)
                  {
                        System.out.println("Am depasit limita la pasul
                        "+i+".");
                  }
            }
      }
}
```

Acesta este rezultatul:

```
5 / 1 este 5
impart la zero la pasul 1!
12 / 3 este 4
1 / 4 este 0
impart la zero la pasul 4!
27 / 5 este 5
```

```
Am depasit limita la pasul 6.
Am depasit limita la pasul 7.
```

In cazul de mai sus, am scris handler-ele astfel ca ele sa trateze si erorile de tipul *ArithmeticException* si cele de tipul *ArrayIndexOutOfBoundsException.* Astfel enumerand blocurile catch unul sub celalalat putem trata mai multe tipuri de exceptii ce pot apare pe parcurs.

Alta remarca ar fi ca in cadrul blocurilor catch regulile domeniilor de vizibilitate al variabilelor si functiilor se mentin. Dupa cum se poate observa folosim *i* din cadrul *for*-ului pentru ca blocul *try catch* este in interiorul acestui *for* si *i* este vizibil tot in interiorul *for*-ului.

Prinderea exceptiilor conform ierahiei

Sa analizam in continuare cealalta metoda de a rezolva cazul de mai sus.

Atunci cand prindem exceptii ca cele de mai sus, este indicat, sa plasam catch-urile unul sub altul, insa nu putem scrie zeci de catch-uri pentru a trata toate erorile posibile.

In Java exista o ierarhie de clase ce se ocupa de exceptii si una care se ocupa de erori. Toate au ca parinte clasa throwable:

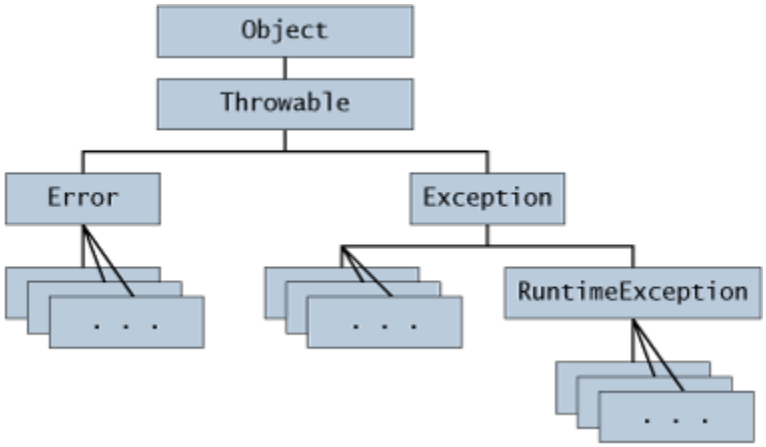


Figura 4.3 Ierarhia claselor de exceptii

Dupa cum se vede in figura clasa *Throwable* este parinte pentru orice clasa de tip *Exception* si *Error*. Asadar, cand scriem codul catch este important sa plasam codul *catch* in ordinea mostenirilor astfel ca urmatoarea ordine este recomandata:

```
try
{
	...
}
catch (RuntimeException rex)
{
	...
}
catch(Exception ex)
{
	...
```

```
	}
	catch(Throwable tex)
	{
		...
	}
```

Sa vedem si un alt exemplu referitor la prinderea exceptiilor matematice:

```
class ExcMultipleDemo
{
	public static void main(String args[])
	{

		int numar[] = { 5, 3, 12, 1, 20, 27, 10, 56 };
		int deimp[] = { 1, 0, 3, 4, 0, 5 };
		for(int i=0; i<numar.length; i++)
		{
			try
			{
				System.out.println(numar[i] + " / " +	deimp[i] +
" este " + numar[i]/deimp[i]);
			}

			catch (ArrayIndexOutOfBoundsException iex)
			{
				System.out.println("Am depasit limita la pasul
"+i+".");
			}
			catch (Throwable tex)
			{
				System.out.println("o alta eroare a aparut.");
			}
		}
	}
}
```

Avantajul acestui cod este ca indiferent de tipul de eroare care apare in blocul try, aceasta va fi tratata. Problema este ca tratarea este una cat mai general aproximata. Eventual putem folosi informatii din parametrul din catch, dupa cum vom vedea in continuare.

Cum aruncam o exceptie?

In exemplele prezentate pana acum, exceptiile au fost generate automat de JVM
Este posibil sa provocam - aruncam o exceptie utilizand **throw**. Forma generala este:

```
throw exceptioObject;
```

In acest caz `exceptioObject` este un obiect de clasa exceptie derivat din **Throwable**.

Mai jos avem un exemplu pentru throw:

```
class ThrowDemo
{
	public static void main(String args[])
	{
		try
		{
			System.out.println("inainte ca exceptia sa apara.");
			throw new ArithmeticException();
		}
		catch (ArithmeticException exc)
		{
			System.out.println("Exceptie prinsa.");
		}
		System.out.println("Dupa blocul try catch.");
	}
}
```

Obiectul de tip `ArithmeticException` a fost creat folosind operatorul **new**. Apoi a fost „aruncat” folosind operatorul **throw**. Mai apoi a fost „prins”, adica transmis ca parametru in blocul **catch**.

Clasa Throwable

Pana in acest moment singurul lucru pe care l-am facut atunci cand am prins erori a fost sa afisam un mesaj corespunzator erorii prinse. Daca profitam de faptul ca toate exceptiile deriva din Throwable, atunci putem folosi functiile acestei clase si anume:

Metoda	**Descriere**
Throwable fillInStackTrace( )	Returneaza un obiect de tip **Throwable** care contine stiva completa cu apelurile efectuate.
String getLocalizedMessage( )	Returneaza o descriere specifica subclasei a erorii.
String getMessage( )	Returneaza descrierea generala a erorii.
void printStackTrace( )	Afiseaza stiva.
void printStackTrace(PrintStream stream)	Trimite continutul stivei unui stream.
String toString( )	Returneaza un sir ce contine descrierea erorii.

Aceste metode se pot apela in blocul catch dupa cum urmeaza:

```
class ExceptionDemo1
{
	public static void genException()
	{
		int nums[] = new int[4];
			System.out.println("Inainte de generarea erorii.");
```

```
                  // Generam o exceptie prin depasirea limitei sirului
                  nums[7] = 10;
                  System.out.println("Eroarea a aparut deja. Instructiunea
aceasta nu va mai fi executata?");


      }
}

class ExceptionDemo2
{
      public static void main(String args[])
      {
            try
            {
                  ExceptionDemo1.genException();
            }
            catch (ArrayIndexOutOfBoundsException exc)
            {
                  System.out.println("Mesajul standard al exceptiei: ");
                  System.out.println(exc);
                  System.out.println("Mesajul exceptiei: ");
                  System.out.println(exc.getMessage());
                  System.out.println("Mesajul local: ");
                  System.out.println(exc.getLocalizedMessage());
                  System.out.println("\nStack trace: ");
                  exc.printStackTrace();
            }
            System.out.println("Dupa blocul try catch.");
      }
}
```

Rezultatul rularii acesui program este unul previzibil:

```
Inainte de generarea erorii.
Mesajul standard al exceptiei:
java.lang.ArrayIndexOutOfBoundsException: 7
Mesajul exceptiei:
7
Mesajul local:
7

Stack trace:
java.lang.ArrayIndexOutOfBoundsException: 7
        at ExceptionDemo1.genException(ex.java:8)
        at ExceptionDemo2.main(ex.java:21)
Dupa blocul try catch.
```

finally

Acest bloc din cadrul declaratiei *try catch* permite executia unor instructiuni inainte de iesirea din blocul *try catch*. De exemplu atunci cand apare o exceptie, se poate efectua un return din functie inainte ca instructiunile de inchidere de fisier sau de conexiuni sa aiba loc. Este recomandat ca aceste tipuri de instructiuni sa le scriem in finally:

```
try
{
      // blocul in care eroarea poate apare
}
catch (ExcepType1 exOb)
{
      // handler pentru ExcepType1
}
finally
{
      //cod care se executa tot timpul
}
```

Mai jos avem o exemplificare a folosirii acestei clauze:

```
class ExFinally
{
      public static void genException(int p)
      {

            int t;
            int nums[] = new int[2];
            System.out.println("Am primit " + p);
            try
            {
                  switch(p)
                  {
                        case 0:
                              t = 6 / p;
                              break;
                        case 1:
                              nums[10] = 9;
                              break;
                        case 2:
                              // return din blocul try
                              return;
                  }
            }
            catch (ArithmeticException exc)
            {

                  System.out.println("Impartire la zero!");
                  return; // return din catch
```

```
            }
            catch (ArrayIndexOutOfBoundsException exc)
            {

                  System.out.println("Limite depasite.");
            }
            finally
            {
                  System.out.println("Parasesc try.");
            }
      }
}
class FinallyDemo
{
      public static void main(String args[])
      {
            for(int i=0; i < 3; i++)
            {
                  ExFinally.genException(i);
                  System.out.println();

            }
      }
}
```

In clasa *FinallyDemo* se apeleaza de trei ori metoda statica *genException()* din clasa *ExFinally*. In functie de parametru se genereaza doua exceptii si in al treilea caz se paraseste blocul *try*. Se observa ca in toate cazurile, indiferent ce *catch* este executat, se intra pe blocul *finally* inainte de a reveni din apelul functiei *genException()*.

```
Am primit 0
Impartire la zero!
Parasesc try.

Am primit 1
Limite depasite.
Parasesc try.

Am primit 2
Parasesc try.
```

In unele cazuri, daca o metoda poate genera o exceptie pe care nu o trateaza, trebuie semnalat acest lucru. Pentru aceasta se foloseste **throws**:

```
tip_de_data nume_metoda(lista_parametrii) throws lista_exceptii
{
      //corpul metodei
}
```

lista_exceptii reprezinta una sau mai multe exceptii care sunt separate prin virgula. Aceste exceptii pot fi generate in interiorul metodei. Exceptiile care sunt subclase ale clasei Error sau al clasei RuntimeException nu trebuie neaparat „declarate” in acea lista. Regula este impusa tuturor celorlalte tipuri de Exceptii.

Iata cateva clase de exceptii, subclase ale lui Exception ce trebuie declarate cu **throws** in metoda apelanta:

CloneNotSupportedException, *IOException*, *NamingException*, *NotBoundException*.

Clasa de mai jos este gresit declarata:

```
import java.io.*;
class ThrowsDemo
{
      public static void writeList(int[] vector)
      {
            PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
            for (int i = 0; i < 10; i++)
            {
                  out.println("valoare de la : " + i + " = " + vector[i]);
            }
            out.close();
      }
}
```

Iar la compilare vom primi urmatoarea eroare:

```
throws.java:6: unreported exception java.io.IOException; must be caught or
decla
red to be thrown
                PrintWriter out = new PrintWriter(new
FileWriter("OutFile.txt"))
;
                                                  ^
1 error
```

Corect este sa declaram ca metoda poate arunca o exceptie de tip IO:

```
public static void writeList(int[] vector) throws IOException
{
      ...
}
```

Iata cateva exceptii, majoritatea derivate din RuntimeException care nu trebuie incluse in lista throws:

Exceptia	**Descriere**
ArithmeticException	Eroare de tip aritmetic, de exemplu impartire la zero.
ArrayIndexOutOfBoundsException	Indexul din sir este in afara limitelor.
ArrayStoreException	Se incearca atribuirea unui element din sir cu o valoare incorecta.
ClassCastException	Cast incorect.
IllegalArgumentException	Parametrul transmis metodei este gresit.
IllegalMonitorStateException	Operatie invalida, de exemplu, asteptarea unui fir de executie neblocat.
IllegalStateException	Aplicatia nu ruleaza in mediu corect, compatibil.
IllegalThreadStateException	Operatia aplicata nu este in concordanta cu starea firului de executie
IndexOutOfBoundsException	Indexul este in afara limitelor
NegativeArraySizeException	Sir creat cu limite negative.
NullPointerException	Folosirea obiectului neinstantiat (egal cu null).
NumberFormatException	Conversia dintr-un string intr-un numar a esuat.
SecurityException	Posibila tentativa asupra securitatii.
StringIndexOutOfBounds	Indexul unui string este in afara limitelor
CloneNotSupportedException	Incercare de a folosi un obiect ce nu implementeaza Cloneable
IllegalAccessException	Acces la clasa nepermis.
InstantiationException	S-a incercat instantierea unei interfete sau clase abstracte
InterruptedException	Un fir de executie a fost intrerupt
NoSuchFieldException	S-a incercat accesarea unui membru care nu exista.
NoSuchMethodException	S-a incercat accesarea unei metode care nu exista.

Desi Java implementeaza multe clase care se ocupa cu tratarea erorilor, mecanismul nu este limitat la genul de liste de mai sus. Putem defini propriile noastre clase ca subclase ale parintelui *Exception*.

Iata un exemplu de creare a unei exceptii customizate:

```
class MyException extends Exception
{
	int n;
	int d;
	MyException(int i, int j)
	{
		n = i;
		d = j;
	}
	public String toString()
	{
		return "Rezultatul " + n + " / " + d + " nu este intreg.";
	}
}
```

```
class CustomExceptionDemo
{
	public static void main(String args[])
	{
		int numar[] = { 3, 6, 30, 44, 21, 50, 16 };
		int deimp[] = { 3, 0, 12 , 2, 5 };
		for(int i=0; i<numar.length; i++)
		{
			try
			{
				if((numar[i]%deimp[i]) != 0)
				throw new MyException(numar[i], deimp[i]);
				System.out.println(numar[i] + " / " + deimp[i] + "
				este " +numar[i]/deimp[i]);
			}
			catch (ArithmeticException exc)
			{
				System.out.println("Impart la zero!");
			}
			catch (ArrayIndexOutOfBoundsException ex)
			{
				System.out.println("Index in afara limitelor.");
			}
			catch (MyException ex)
			{
				System.out.println(ex);
			}
		}
	}
}
```

In primul rand clasa trebuie sa mosteneasca Exception. In al doilea rand aruncarea exceptiei se face prin apelul throws cu un obiect de clasa customizata. Pentru crearea unui obiect de acest tip de exceptie s-a folosit un constructor cu doi parametrii de tip int. Dupa ce exceptia este aruncata se poate folosi obiectul *ex* care este de tip *MyException*, ca orice obiect din Java.

I/O in Java

Sistemul de tratare al intrarilor si iesirilor si al lucrului cu siruri de data in Java este extrem de variat. In cele ce urmeaza vom incerca sa surprindem conceptele, practic sa vedem varful iceberg-ului, pentru ca mai tarziu sa intram in detalii anumite aspecte legate de IO.

Stream in Java

Operatiuniile de scriere/citire in Java se realizeaza cu ajutorul stream-urilor. Un stream este o abstractie a datelor concrete conectat la un sistem de I/O (fisier, memorie, imprimanta etc).

In general va exista o sursa de la care vom citi si atunci vorbim de input stream si o sursa catre care scriem si atunci vorbim de output stream:

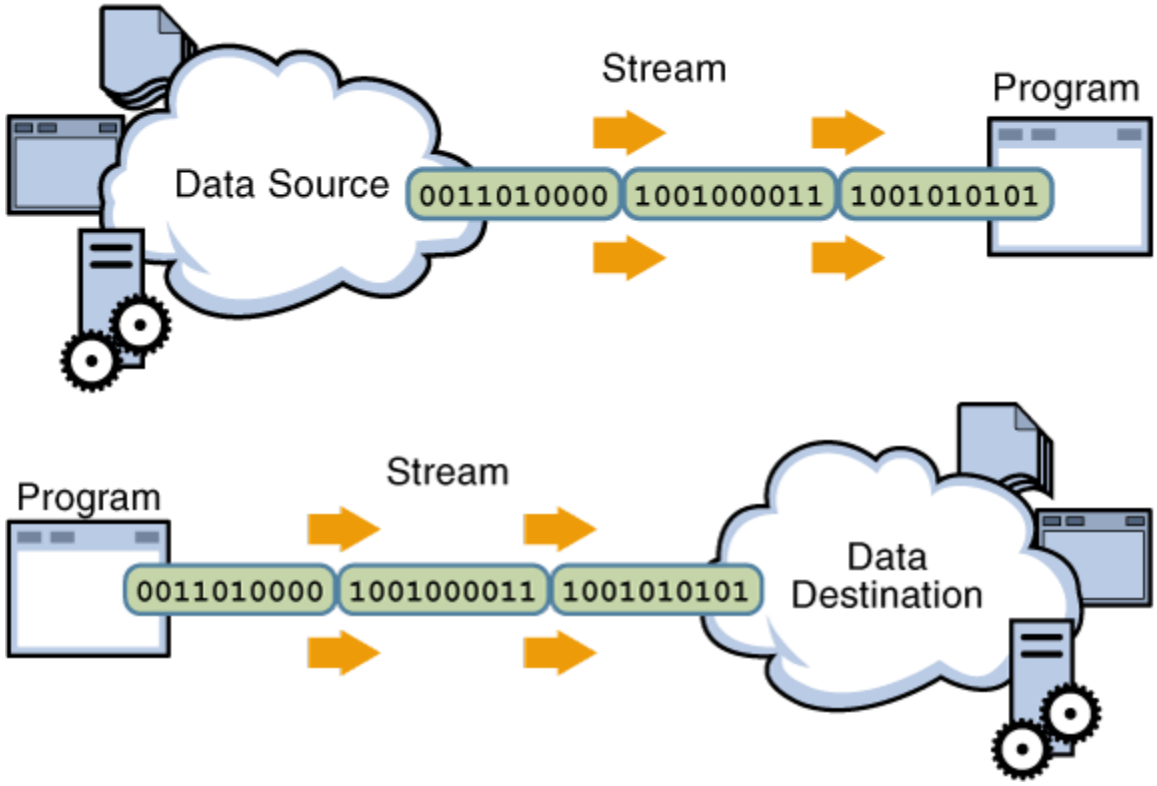


Fig 4.4 Stream-uri I/O

Byte Stream si Character Stream

In Java exista doua versiuni de stream: byte si caracter. Byte stream-ul este un wrapper peste un sir de byte. Acesta se foloseste la scrierea si citirea datelor binare si la transmiterea, citirea de fisiere. Stream-urile de caractere sunt destinate lucrului cu texte si folosesc Unicode pentru a putea fi internationalizate.

Clasele Byte Stream sunt definite prin doua ierarhii de clase: *InputStream* si *OutputStream* din care deriva o serie de clase pentru lucrul cu diverse dispozitive.

Clasele destinate lucrului cu text, si anume Character Stream, sunt *Reader* si *Writer*. Analog din acestea deriva o serie de subclase avand particularitatile lor.

Exista stream-uri predefinite in Java pentru lucru cu intrarea standard : **System.in**, iesirea standard **System.out** si eroarea standard **System.err**.

Byte Stream

Ierarhia tipurilor de data byte stream este prezentata mai jos:

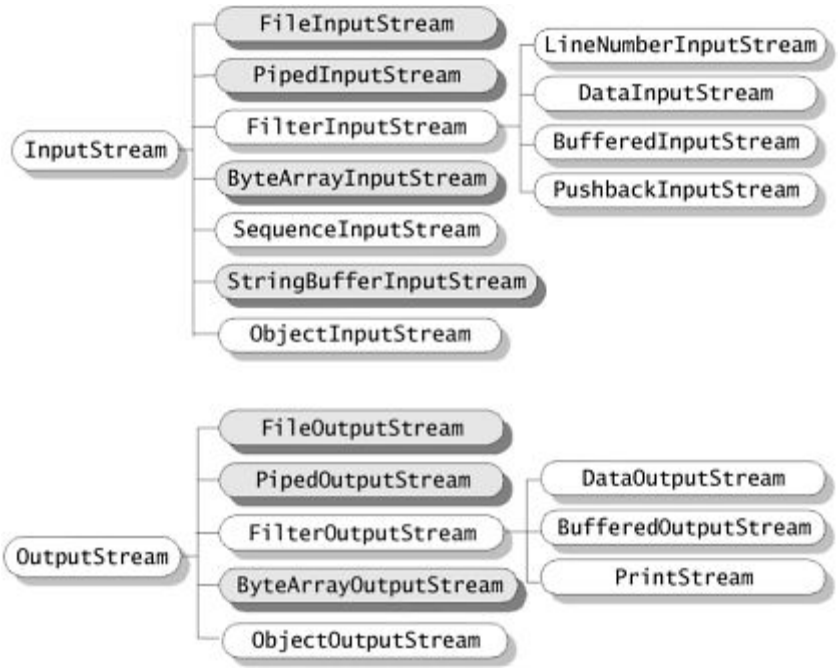


Figura 4.5 Ierarhia claselor byte stream

In tabelul de mai jos avem explicatiile pentru aceste clase:

Clasa	**Scop**
BufferedInputStream	input stream ca buffer
BufferedOutputStream	output stream ca buffer
ByteArrayInputStream	Input stream care citeste din sir de byte
ByteArrayOutputStream	Output stream ce scrie in sir de byte
DataInputStream	Un input stream ce contine metode pentru citirea tipurilor standard de data din Java
DataOutputStream	Un output stream ce contine metode pentru scrierea tipurilor standard de data din Java
FileInputStream	Input stream care citeste dintr-un fisier
FileOutputStream	Output stream care scrie intr-un fisier
FilterInputStream	Implementeaza InputStream

FilterOutputStream	Implementeaza OutputStream
InputStream	Clasa abstracata ce descrie input stream
ObjectInputStream	Input stream pentru obiecte
ObjectOutputStream	Output stream pentru obiecte
OutputStream	Clasa abstracata ce descrie output stream
PipedInputStream	pipe pentru intrari
PipedOutputStream	pipe pentru iesiri
PrintStream	Output stream ce poate apela **print( )** si **println( )**
PushbackInputStream	Input stream care permite ca byte-tii sa fie returnati dintr-un stream
SequenceInputStream	Input stream ce este o combinatie de mai multe stream-uri de intrari ce vor fi citite secvential unul dupa celalalt

.

Iata si une exemplu de folosire al unora dintre clasele mai sus mentionate:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("filei.txt");
            out = new FileOutputStream("fileo.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }

        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

In exemplul de mai sus este folosita functia *read()* care are ca scop citirea byte cu byte a intrarii. De altfel, clasa InputStream defineste o serie de metode ce sunt mostenite si de copii sai:

Metoda	Descriere
int available( )	Returneza numbarul de bytes disponibili de la intrare.
void close( )	Inchide sursa. Alte incercari de a citi dupa, vor genera IOException.
void mark(int numBytes)	Pune un semn la byte-ul curent care va ramane valid pana ce vom fi citit *numBytes*.
boolean markSupported( )	Returneaza true daca stream-ul suporta sistemul de marcare descris.
int read( )	Returneaza un integer ce reprezinta byte-ul urmator din stream sau –1 daca s-a intalnit sfarsit de fisier.
int read(byte buffer[ ])	Se incearca citirea unui buffer de lungimea celui specificat in paranteze si returneaza numarul de bytes citit sau -1 daca s-a intalnit sfarsit de fisier.
int read(byte buffer[ ], int offset,int numBytes)	Se incearca citirea unui buffer de lungime numBytes, in buffer incepand cu offset, si returneaza numarul de bytes cititi sau -1 daca am ajuns la sfarsitul fisierului.
void reset( )	Reseteaza acel semn stabilit de mark.
long skip(long numBytes)	Ignora numBytes bytes din intrare si returneaza numarul de bytes ignorat.

Mai jos avem descrise metodele specifice unui OutputStream:

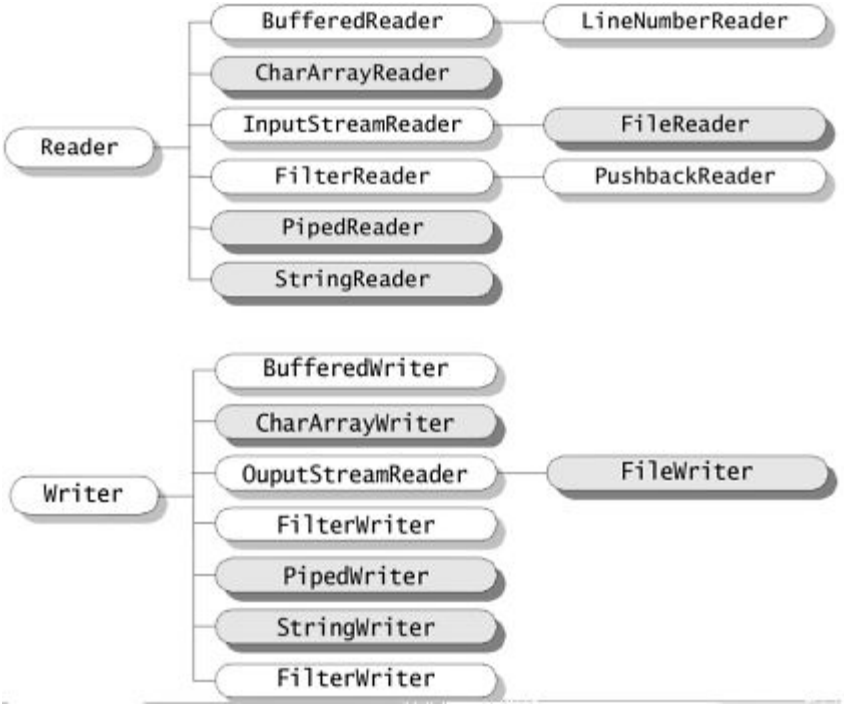
Metoda	Descriere
void close( )	Inchide stream-ul de scriere catre iesire, o alta scriere generand **IOException**.
void flush( )	Forteaza scrierea din buffer la iesire, curatind bufferul.
void write(int b)	Scrie un byte la iesire. Parametru este un int.
void write(byte buffer[ ])	Scrie un sir de bytes la iesire.
void write(byte buffer[ ], int offset, int numBytes)	Scrie o portiune dintr-un buffer, delimitat de offset si numBytes.

Character Stream

Platforma Java permite utilizarea carcterelor folosind conventii Unicode. Stream-urile carcater sunt automat transpuse intr-un format intern, care poate fi ulterior convertit la diverse formate (depinde de locul unde va fi instalata aplicatia).

Pentru majoritatea aplicatiilor, stream-urile de carctere I/O, sunt la fel de simple ca cele byte si regulile sunt aceleasi. Situatia se poate complica daca programul va fi unul international.

Toate clasele de acest gen deriva din Reader sau Writer. Iata mai jos ierarhia acestora:



Mai jos este un program pentru exemplificarea utilizarii acestor clase:

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

class CopyCharacters
{
    public static void main(String[] args) throws IOException
{
        FileReader inputStream = null;
        FileWriter outputStream = null;

        try
      {
            inputStream = new FileReader("filei.txt");
            outputStream = new FileWriter("fileo.txt");

            int c;
            while ((c = inputStream.read()) != -1)
            {
                outputStream.write(c);
            }
        }
        finally
        {
            if (inputStream != null)
```

```
            {
                inputStream.close();
            }
            if (outputStream != null)
            {
                outputStream.close();
            }
        }
    }
}
```

Metodele suportate de aceste clase sunt aceleasi ca si la byte streams cu deosebirea ca aceste metode lucreaza cu char si nu byte. De exemplu in clasa Reader, iata cateva metode ce difera sau sunt noi:

int read(CharBuffer buffer)	Se incearca citirea unui buffer de lungimea celui specificat in paranteze si returneaza numarul de bytes citit sau -1 daca s-a intalnit sfarsit de fisier. CharBuffer este o clasa ce incapsuleaza o secventa de caractere asa cum face String.
int read(char buffer[ ])	Acelasi ca in cazul byte:de data aceasta se citesc caractere

In cazul clasei Writer exista cateva metode noi:

Writer append(char ch)	scrie la sfarsitul stream-ului caracterul *ch*.
Writer append(CharSequence chars)	scrie secventa chars la sfarsitul stream-ului. Returneaza o referinta a stream-ului ce invoca metoda. CharSequence este o interfata ce defineste operatii read-only pe o secventa de caractere.
Writer append(CharSequence chars, int begin, int end)	scrie secventa chars la sfarsitul stream-ului delimitata de begin si end.

Stream-uri salvate in buffer-e

Majoritatea exemplelor nu controlau buffer-ul in care se salveaza datele inainte de a fi afisate sau citite. Pentru a optimiza accesul la disc, retea sau oricare sursa implicata in schimbul de date se poate folosi un mecanism Java si anume Buffered Streams. Acestea sunt siruri de date din memorie in care se plaseaza date controlat.

Clasele pentru stream-uri byte sunt BufferedInputStream si BufferedOutputStream in timp ce pentru stream-urile caracter exista BufferedReader si BufferedWriter.

Iata si un exemplu de folosire al acestor clase:

```
import java.io.*;
class BufferDemo
{
      public static void main(String args[]) throws Exception
      {
            FileReader fr = new FileReader("filei.txt");
            BufferedReader br = new BufferedReader(fr);
            String s;
            while((s = br.readLine()) != null)
            {
                  System.out.println(s);
            }
            fr.close();
      }
}
```

Evident scopurile pot fi multiple, si in general se va folosi pentru citirea controlata in buffer-e de o anumita dimensiune.