Lecţia 11

Elemente de programare concurentă

Multe sisteme software reale, scrise în Java sau în alte limbaje de programare, trebuie să trateze mai multe evenimente simultan. În această lecţie vom vedea cum anume putem crea astfel de programe în Java, ce probleme specifice pot apare în astfel de programe şi ce mecanisme de limbaj pune la dispoziţie Java pentru a evita aceste probleme.

11.1 Concurenţa

În teoria programării orientate pe obiecte concurenţa se defineşte în felul următor:

Definiţie 4 *Concurenţa este proprietatea care distinge un obiect activ de un obiect pasiv.*

Dar ce înseamnă un obiect activ şi ce înseamnă un obiect pasiv? Un obiect activ este un obiect care "face ceva" fără ca alt obiect exterior lui să acţioneze asupra sa prin apelarea unei metode. Un obiect este pasiv dacă el stă şi nu "face nimic" dacă asupra sa nu acţionează un alt obiect exterior lui.

În lumea ce ne înconjoară există o grămadă de obiecte pasive şi tot atâtea obiecte active. Un exemplu de obiect real pasiv este arcul unui indian. Arcul "face ceva", mai exact lansează săgeţi, doar când indianul acţionează asupra lui. Arcul nu lansează săgeţi când are el chef. Un obiect real activ e ceasul de mână. El schimbă indicaţia orară tot timpul fără acţiunea unui alt obiect exterior. E adevărat că un obiect exterior îl poate regla dar asta nu-l face obiect pasiv. Dacă ar fi pasiv atunci el ar trebui reglat din exterior la trecerea fiecărei secunde pentru a arăta ora exactă. Nu aş vrea un asemenea ceas de mână.

De ce avem nevoie de obiecte active? În orice sistem software trebuie să existe cel

puţin un obiect activ, altfel sistemul în ansamblul său nu ar "face nimic". Nu ar exista nimic în acel sistem care să manipuleze obiectele pasive ce-l compun pentru ca acestea să "facă ceva". Dacă într-un sistem avem mai multe obiecte active este posibil ca mai multe obiecte din acel sistem, inclusiv pasive, să "facă ceva" în acelaşi timp, iar sistemul în ansamblul său va "face" mai multe lucruri simultan. Spunem în astfel de situaţii că avem un sistem sau program concurent.

Un indian este evident un obiect real activ. Doi indieni pot acţiona simultan două arcuri distincte şi prin urmare aceste două obiecte reale pasive pot lansa săgeţi în acelaşi timp!

11.2 Ce este un fir de execuţie?

În secţiunea anterioară am vorbit despre obiecte active şi pasive. Până acum noi am definit, creat şi manipulat o sumedenie de obiecte în Java. Erau ele pasive sau active?

Toate obiectele definite de noi până acum erau pasive: pentru ca ele să "facă ceva" trebuia să acţionăm asupra lor din exterior prin apelarea unei metode. După cum am spus la început, un obiect activ "face ceva" fără a se acţiona din exteriorul său (tipăreşte mesaje pe ecran, apelează metode ale altor obiecte, etc.). În aceste condiţii spunem că un obiect activ are propriul său *fir de execuţie*. Noţiunea de fir de execuţie îşi are originea în domeniul sistemelor de operare. În continuare vom discuta principial ce înseamnă un fir de execuţie din punct de vedere fizic.

Important

Discuţia ce urmează nu corespunde perfect realităţii dar e utilă pentru înţelegerea conceptelor.

Când un program Java e lansat în execuţie maşina virtuală îi alocă o porţiune de memorie. Această porţiune e împărţită în trei zone: o zonă de date (heap), o zonă pentru stive şi o zonă pentru instrucţiunile programului. În timp ce programul se execută, putem să ne imaginăm că instrucţiuni din zona de instrucţiuni "curg" înspre procesor, acesta executându-le succesiv. Acest flux de instrucţiuni (mai exact copii ale instrucţiunilor deoarece ele nu pleacă din memorie) ce "curge" dinspre zona de instrucţiuni spre procesor se numeşte *fir de execuţie*. În cadrul unui program Java este posibil ca la un moment dat să avem două sau mai multe fluxuri de instrucţiuni ce "curg" dinspre aceeaşi zonă de instrucţiuni spre procesor. Spunem că acel program se execută în mai multe fire de execuţie simultan. Efectul pe care-l percepe programatorul este că mai multe părţi din acelaşi program (nu neapărat distincte din punct de vedere fizic) se execută în acelaşi timp.

Dar cum poate un singur procesor să primească şi să execute în acelaşi timp două fluxuri de instrucţiuni? Răspunsul e că nu poate. Este însă posibil ca fluxurile de

instrucţiuni să "curgă" spre două procesoare diferite în cazul sistemelor de calcul cu mai multe procesoare. Dacă nu avem decât un singur procesor atunci maşina virtuală aplică un algoritm de *time-slicing*: o perioadă de timp sau o cuantă de timp procesorul execută instrucţiuni dintr-un flux, apoi o altă perioadă de timp din altul, apoi din altul, ş.a.m.d. Practic procesorul e dat de maşina virtuală o perioadă de timp fiecărui fir de execuţie individual creînd astfel iluzia unei execuţii simultane a mai multor părţi din acelaşi program. Când unui fir i se ia procesorul, el va aştepta până îl va primi înapoi şi doar atunci îşi va continua execuţia. De unde? Exact de acolo de unde a lăsat-o când i s-a luat procesorul.

Legat de firele de execuţie în care se execută la un moment dat un program, trebuie subliniate următoarele aspecte:

- toate firele de execuţie văd acelaşi heap. Java alocă toate obiectele în heap, deci toate firele au acces la toate obiectele.
- fiecare fir are propria sa stivă. Variabilele locale şi parametrii unei metode se alocă în stivă, deci fiecare fir are propriile valori pentru variabilele locale şi pentru parametrii metodelor pe care le execută.
- corecta funcţionare a unui program concurent nu are voie să se bazeze pe presupuneri legate de viteza relativă de execuţie a firelor sale. Argumente de genul "ştiu eu că firul ăsta e mai rapid ca celălalt" sau "ştiu eu că obiectul ăsta nu e accesat niciodată simultan de două fire de execuţie diferite" sunt interzise.

11.3 Utilizarea firelor de execuţie în Java

11.3.1 Definirea, crearea şi declanşarea firelor de execuţie

În Java, definirea şi crearea unui obiect activ se traduce prin definirea şi crearea unui obiect fir de execuţie căruia îi va corespunde un unic fir de execuţie fizic din maşina virtuală. Pentru a se putea executa, un program Java are nevoie de cel puţin un fir de execuţie în maşina virtuală (altfel instrucţiunile sale nu ar avea cum ajunge la procesor). Principial, am putea spune că în orice program Java aflat în execuţie există un obiect fir de execuţie: el nu e definit şi nici creat explicit de programator, dar el există, fiind creat automat de maşina virtuală Java la pornirea programului având rolul de a apela metoda *main*.

Şi un programator poate defini şi crea obiecte fire de execuţie şi deci poate scrie programe ce se execută în mai multe fire de execuţie fizice ale maşinii virtuale. Există două variante pe care le prezentăm mai jos. A doua porţiune de cod ne arată cum anume creăm astfel de obiecte (corespunzător celor două variante) şi cum anume declanşăm funcţionarea lor (mai exact cum dăm drumul firelor lor fizice de execuţie).

```
//Varianta 1
class FirUnu extends Thread {
    public void run() {
    ...
    }
}

//Varianta 2
class FirDoi implements Runnable {
    public void run() {
    ...
    }
}
```

```
//Varianta 1
FirUnu o = new FirUnu();
o.start();

//Varianta 2
Thread o = new Thread(new FirDoi());
o.start();
```

În prima variantă definirea unui obiect fir de execuţie se face printr-o clasă ce moşteneşte clasa predefinită *Thread*. Mai mult, se suprascrie metoda *run* moştenită de la clasa *Thread*. Ea va avea rolul unui fel de metodă *main*, în sensul că de acolo începe să se execute respectivul fir de execuţie. Crearea unui obiect fir se reduce în această variantă la crearea unei instanţe a respectivei clase. Pentru declanşarea execuţiei firului trebuie să apelăm metoda sa *start*, moştenită de la clasa *Thread*.

În cea de-a doua variantă definirea unui obiect fir de execuţie se face printr-o clasă ce implementează interfaţa *Runnable* şi care implementează metoda *run* ce are acelaşi rol ca în cazul primei variante. Crearea unui obiect fir de execuţie se va face în acest caz prin crearea unei instanţe a clasei *Thread* utilizând un constructor ce primeşte ca parametru un obiect *Runnable*. În cazul nostru el va primi ca parametru o instanţă a clasei despre care am vorbit înainte. Pornirea firului se face utilizând tot metoda *start*.

11.3.2 Cooperarea firelor de execuţie

O caracteristică importantă a firelor de execuţie este că ele văd acelaşi heap. Acest lucru poate duce la multe probleme în cazul în care un obiect oarecare (care e o resursă comună pentru toate firele) este accesat din două fire de execuţie diferite. Vom considera un exemplu ipotetic pentru a vedea ce se poate întâmpla.

Este evident că în momentul în care doi indieni vor să utilizeze **acelaşi** arc simultan vor apare ceva probleme.

```
class Patrat {

    private int latime,inaltime;

    public void set(int latura) {
        /*1*/latime = latura;
        /*2*/inaltime = latura;
        /*3*/if(latime == inaltime) {
            System.out.println("Patratul are latura " + latime);
        }
        /*4*/else System.out.println("ERROR");
    }
}
```

Să presupunem clasa de mai sus ce modelează un pătrat. Exemplul are doar un rol ilustrativ: cine ar folosi pentru a memora latura unui pătrat două câmpuri despre care ştim că sunt egale tot timpul? Să presupunem însă că se întâmplă acest lucru şi că respectiva clasă este utilizată într-un program concurent. Evident, în programele în care avem un singur fir de execuţie nu apare nici o problemă: niciodată nu se va tipări pe ecran mesajul ERROR. La fel se întâmplă şi într-un program concurent dacă nici un obiect pătrat nu e accesat simultan din mai multe fire de execuţie distincte. Probleme apar când două (sau mai multe) fire de execuţie acţionează simultan asupra ACELUIAŞI obiect pătrat. De exemplu, un fir apelează metoda *set* cu parametrul actual 1 şi un alt fir apelează metoda *set* cu parametrul actual 2 (dar pentru ACELAŞI obiect). Se poate întâmpla următorul scenariu:

- primul fir execută apelul şi apoi linia 1 din program, dând valoarea 1 câmpului lăţime.
- după ce se execută linia 1 maşina virtuală hotărăşte că trebuie să dea procesorul altui fir, opreşte pentru moment execuţia primului fir şi porneşte execuţia celui de-al doilea.
- al doilea fir execută apelul metodei *set* şi apoi instrucţiunile 1 şi 2 dând aceeaşi valoare (2) ambelor câmpuri; apoi execută testul, totul e ok, şi iese din metodă.
- la un moment dat maşina virtuală dă iar procesorul primului fir şi acesta reîncepe execuţia cu instrucţiunea numărul 2. Cum am spus că fiecare fir are propria stivă, parametrul *latura* are valoarea 1 în acest fir; prin urmare înălţimea va lua valoarea

1, după care se execută instrucţiunea 3. SURPRIZĂ: laturile nu mai sunt egale şi prin urmare se tipăreşte pe ecran ERROR.

Acelaşi scenariu poate apare şi dacă avem mai multe procesoare, nu numai când se foloseşte algoritmul de time-slicing.

Totuşi, intenţia programatorului a fost de a ţine cele două laturi tot timpul egale. Unde a greşit? Problema se datorează faptului că programatorul, deşi ştia că asupra aceluiaşi obiect ar putea acţiona mai multe fire de execuţie, a presupus implicit că odată ce un fir începe să execute metoda *set* pentru un obiect, el o va termina înainte ca un alt fir să înceapă să execute şi el metoda *set* pentru acelaşi obiect. Cu alte cuvinte, a neglijat faptul că pentru ca programul să meargă corect, nu e voie să se facă vreo presupunere legată de viteza relativă de execuţie a mai multor fire.

În cazul exemplului de mai sus se spune că programatorul a introdus în sistem o *condiţie de cursă*. Astfel de condiţii nu au voie să apară în sistemele concurente. Menţionăm că nu toate condiţiile de cursă se vor elimina aşa de uşor ca în exemplul nostru.

Exemplul de mai sus ne arată că atunci când avem mai multe fire de execuţie pot apare probleme extrem de subtile. Pentru eliminarea lor este necesar ca firele de execuţie să poată coopera cumva între ele. În Java există două mecanisme de cooperare: mecanismul de excludere mutuală şi mecanismul de cooperare prin condiţii.

Imaginaţi-vă o orchestră simfonică. Fiecare membru al orchestrei e un obiect activ care cântă partea sa din compoziţia interpretată. Dacă fiecare ar cânta după bunul său plac ar rezulta un haos. Dar nu se întâmplă acest lucru pentru că aceste "fire de execuţie" cooperează între ele într-o manieră impusă de dirijor. Din păcate în sistemele software aflate la execuţie nu există un astfel de dirijor. Programatorul joacă rolul dirijorului în sensul că el utilizează anumite mecanisme de limbaj ce impun firelor un mod clar de cooperare.

Mecanismul de excludere mutuală

Excluderea mutuală se traduce informal prin faptul că la un moment dat un obiect poate fi manipulat de un singur fir de execuţie şi numai de unul. Cum specificăm acest lucru? O posibilitate constă în utilizarea modificatorului de acces la metode *synchronized*. Rolul său e simplu: în timpul în care un fir execută instrucţiunile unei metode *synchronized* pentru un obiect, nici un alt fir nu poate executa o metodă declarată *synchronized* pentru ACELAŞI obiect.

Important Excluderea mutuală se poate obţine şi prin utilizarea *blocurilor de sincronizare*. Despre ele nu vom vorbi aici, dar principiul e foarte asemănător.

Cine impune această regulă? Principial vom spune că maşina virtuală Java. Ea are grijă singură, fără nici o intervenţie exterioară, ca această regulă să fie respectată. Cum? Destul de simplu. În momentul în care un fir de execuţie apelează o metodă sincronizată pentru un obiect se verifică dacă obiectul respectiv se află în starea "liber". Dacă da, obiectul e trecut în starea "ocupat" şi firul începe să execute metoda, iar când firul termină execuţia metodei obiectul revine în starea "liber". Dacă nu e "liber" când s-a efectuat apelul, înseamnă că există un alt fir ce execută o metodă sincronizată pentru acelaşi obiect (mai exact, un alt fir a trecut obiectul în starea "ocupat" apelând o metodă sincronizată sau utilizând un bloc de sincronizare). Într-o astfel de situaţie firul va aştepta până când obiectul trece în starea "liber".

Dacă observaţi cu atenţie, este acelaşi mod de cooperare pe care-l aplică oamenii în multe situaţii. Oamenii (firele de execuţie) stau la coadă la un ghişeu şi nu năvălesc toţi deodată să vorbească cu unica persoană de la ghişeu (obiectul pe care toate firele vor să-l acceseze). Când un client discută cu persoana de la ghişeu aceasta e ocupată. Când clientul a terminat de discutat, el pleacă iar ghişeul devine liber şi altcineva poate să-l ocupe.

Să reluăm acum exemplul din secţiunea anterioară declarând metoda *set* ca având un acces sincronizat.

```
class Patrat {

    private int latime,inaltime;

    public synchronized void set(int latura) {
        /*1*/latime = latura;
        /*2*/inaltime = latura;
        /*3*/if(latime == inaltime) {
            System.out.println("Patratul are latura " + latime);
             }
        /*4*/else System.out.println("ERROR");
    }
}
```

În această situaţie nu mai există posibilitatea ca pe ecran să se afişeze ERROR indiferent cât de multe fire de execuţie accesează simultan un acelaşi obiect pătrat. Pentru a înţelege motivul să reluăm scenariul de execuţie prezentat anterior: un fir apelează metoda *set* cu parametrul actual 1 şi un alt fir apelează metoda *set* cu parametrul actual 2 (dar pentru ACELAŞI obiect).

- primul fir execută apelul şi apoi execută linia 1 din program, dând valoarea 1

câmpului lăţime.

- după ce se execută linia 1 maşina virtuală hotărăşte că trebuie să dea procesorul altui fir, opreşte pentru moment execuţia primului fir şi porneşte execuţia celui de-al doilea.
- al doilea fir ar vrea să execute apelul dar NU poate deoarece obiectul e "ocupat", primul fir aflându-se în execuţia metodei *set* pentru acel obiect. Ca urmare, firul aşteaptă. Maşina virtuală îşi dă seama că firul nu poate continua şi dă procesorul altui fir.
- la un moment dat maşina virtuală dă iar procesorul primului fir iar acesta reîncepe execuţia cu instrucţiunea numărul 2. Cum am spus că fiecare fir are propria stivă, parametrul latura are valoarea 1 în acest fir; prin urmare înălţimea va lua valoarea 1, după care se execută instrucţiunea 3 fără nici o surpriză, laturile fiind egale. Firul termină apoi execuţia metodei iar obiectul va fi trecut în starea "liber". Când procesorul va ajunge iar la al doilea fir, acesta va putea continua (evident dacă obiectul nostru nu a trecut între timp în starea "ocupat" datorită unui apel la *set* din alt fir ce s-a executat înainte ca al doilea fir să primească procesorul).

Cooperarea prin condiţii

Există situaţii în care excluderea mutuală este necesară pentru rezolvarea unei probleme de concurenţă, dar nu e şi suficientă. Astfel de situaţii pot apare atunci când un fir execută o metodă sincronizată (deci obiectul apelat e "ocupat") dar el nu poate să termine execuţia metodei pentru că nu s-a îndeplinit o anumită condiţie. Dacă acea condiţie poate fi îndeplinită doar când un alt fir ar apela o metodă sincronizată a aceluiaşi obiect, situaţia e fără ieşire: obiectul e ţinut "ocupat" de firul ce aşteaptă apariţia condiţiei, iar firul ce vrea să apeleze o metodă sincronizată a aceluiaşi obiect pentru îndeplinirea condiţiei, pur şi simplu nu poate face acest lucru (obiectul fiind "ocupat" iar metoda e sincronizată). Aceasta e o situaţie de *impas (deadlock)*. Astfel de situaţii nu au voie să apară în sisteme software concurente pentru că le blochează.

O astfel de cooperare aplică şi oamenii în anumite situaţii. Să presupunem că mai multe persoane (fire de execuţie) iau masa la un restaurant. Pe masă există o singură sticlă de sos de roşii. Evident e necesar un mecanism de excludere mutuală pentru utilizarea acelei sticle: doar o singură persoană o poate utiliza la un moment dat. Dar ce se întâmplă când o persoană constată că sticla e goală? Ea deţine sticla la momentul respectiv, dar nu o poate utiliza. Ce se întâmplă? De obicei intervine un alt fir de execuţie, mai exact chelnerul, care umple respectiva sticlă: o preia de la persoana care o deţine, o umple, după care o dă înapoi respectivei persoane. Condiţia fiind îndeplinită, mai exact sticla conţine sos, persoana poate să o utilizeze. Şi în cazuri reale ar putea apare o situaţie de impas: persoana ce deţine sticla goală e atât de încăpăţânată, încât nu vrea să dea sticla chelnerului pentru

a o umple.

Mecanismul de cooperare prin condiţii se bazează pe o pereche de două metode definite în clasa *Object* (datorită acestui lucru toate obiectele dintr-un program au aceste metode). Un lucru important ce trebuie spus încă de la început este că un fir de execuţie care apelează aceste metode pentru un obiect, trebuie să fie singurul fir care poate accesa acel obiect la momentul respectiv. Altfel spus, obiectul trebuie să fie "ocupat" şi să fi trecut în această stare datorită execuţiei firului ce apelează aceste metode. Altfel se va produce o excepţie. În cazurile studiate de noi în această lecţie, un apel la aceste metode e sigur dacă el apare în metode sincronizate şi are ca obiect ţintă obiectul *this*. Situaţiile mai complicate nu vor fi prezentate în această lecţie. Motivul acestei constrângeri rezultă din tabelul ce explică semantica metodelor *wait()* şi *notifyAll()*.

Prototip	Descriere
wait()	Când un fir de execuţie apelează această metodă pentru un obiect, firul va fi pus în aşteptare. Aceasta înseamnă că nu se revine din apel, că ceva ţine firul în interiorul metodei wait(). Acest apel este utilizat pentru "blocarea" unui fir până la îndeplinirea condiţiei de continuare. În timp ce un fir aşteaptă în metoda wait() apelată pentru un obiect, obiectul receptor va trece temporar în starea "liber".
notifyAll()	Când un fir de execuţie apelează această metodă pentru un obiect, TOATE firele de execuţie ce sunt "blocate" în acel moment în metoda wait() a ACELUIAŞI OBIECT sunt deblocate şi pot reveni din apelul respectivei metode. Acest apel este utilizat pentru a anunţa TOATE firele care aşteaptă îndeplinirea condiţiei de continuare că această condiţie a fost satisfăcută.

Tabelul 11.1: Metode utilizate în mecanismul de cooperare prin condiţii.

Se consideră un program simplu compus din două fire de execuţie. Un fir pune numere într-un obiect container, iar alt fir ia numere din ACELAŞI obiect container. Clasa *Container* va avea două metode: una de introducere a unui număr în container, iar alta de extragere a unui număr din container. Containerul poate conţine la un moment dat maxim un număr. Se pun condiţiile: nu putem pune un număr în container dacă containerul e plin şi nu putem scoate un număr din container dacă containerul e gol. O primă variantă incorectă de definire a clasei *Container* e următoarea.

```
class Container {

    private int numar;
    private boolean existaNumar = false;
```

```
    public synchronized void pune(int l) {
        while(existaNumar);  //astept sa fie containerul gol
            numar = l;
        existaNumar = true;  //am pus un numar deci nu mai
                             //e loc de altul
    }

    public synchronized int extrage() {
        while(!existaNumar); //astept sa fie containerul umplut
        existaNumar = false; //scot numarul, deci nu mai e nimic
                             //in container
        return numar;
    }
}
```

De ce e această variantă incorectă? Pentru că poate apare o situaţie de impas (deadlock). Primul fir poate pune un număr în containerul nostru. Prin urmare *existaNumar* devine *true*. Din păcate nu putem să garantăm că al doilea fir este suficient de rapid să vină şi să extragă numărul din container. Prin urmare este posibil ca tot primul fir să mai vrea să pună alt număr în container, dar cum acesta e plin, el stă şi aşteaptă în ciclul *while* până se îndeplineşte condiţia de continuare: adică până al doilea fir ia numărul din container: mai exact va aştepta pe vecie. Motivul? Al doilea fir nu are cum extrage numărul pentru că metoda *extrage* e sincronizată!!! Aşadar avem un impas. Şi dacă programul ăsta controlează ceva de genul unei centrale nucleare, cu siguranţă nu ar fi prea plăcută această situaţie.

O soluţie naivă ar fi să spunem: nici o problemă, facem ca metoda *extrage* să nu fie sincronizată. Dacă am face aşa ceva am da direct din lac în puţ! Continuând situaţia ipotetică de mai sus s-ar putea ca firul doi să înceapă să execute metoda *extrage* şi exact după ce pune *existaMumar* pe *false* maşina virtuală să-i ia procesorul şi să-l dea primului fir. Grav! Primul fir pune alt număr în container! La acordarea procesorului celui de-al doilea fir se va continua de la *return* şi iată cum s-a pierdut un număr! Şi iarăşi ar putea să iasă rău dacă programul controlează ceva dintr-o centrală nucleară!

Soluţia pentru rezolvarea acestei probleme e utilizarea cooperării prin condiţii. Codul corect e prezentat mai jos. Mai mult, codul funcţionează bine şi dacă există mai multe fire care pun numere în acelaşi container şi mai multe fire ce extrag numere din acelaşi container. Vom prezenta acum şi codul ce defineşte cele două fire de execuţie.

```
class FirPuneNumere extends Thread {

    private Container c;
```

```
    public FirPuneNumere(Container c) {
        this.c = c;
    }

    public void run() {
        int i = 0;
        while(true) { i++; c.pune(i % 1000);}
    }
}

class FirExtrageNumere extends Thread {

    private Container c;

    public FirExtrageNumere(Container c) {
        this.c = c;
    }

    public void run() {
        while(true) { System.out.println(c.extrage());}
    }
}

class ProgramTest {

    public static void main(String[] argv) {
        Container c = new Container();
        new FirPuneNumere(c).start();
        new FirExtrageNumere(c).start();
    }

}

class Container {

    private int element;
    private boolean existaNumar = false;

    public synchronized void pune(int l) {
        while(existaNumar) {
            try { wait();  //astept indeplinirea conditiei "container gol"
            } catch(InterruptedException e) {...}
        }
        element = l;
        existaNumar = true;  //am pus un numar deci nu mai
                             //e loc de altul
```

```
        notifyAll(); //Anunta indeplinirea conditiei "container plin"
                     //Toate firele ce sunt blocate in acest moment in wait-
                     //ul din metoda extrage se pregatesc sa revina din el
    }

    public synchronized int extrage() {
        while(!existaNumar) {
            try { wait();  //astept indeplinirea conditiei "container plin"
            } catch(InterruptedException e) {...}
        }
        existaNumar = false; //scot numarul, deci nu mai e nimic
                             //in container
        notifyAll(); //Anunta indeplinirea conditiei "container gol"
                     //Toate firele ce asteapta in acest moment la wait-ul
                     //din metoda pune se pregatesc sa revina din el

        return element;
    }
}
```

Pentru a înţelege funcţionarea metodelor *wait* şi *notifyAll* să vedem mai întâi ce ne încurca în prima versiune a codului. Ne încurca faptul că primul fir stătea în ciclul *while* în aşteptarea golirii containerului. Cum metoda *extrage* e sincronizată, el ţinea obiectul container "ocupat" şi nu-l lăsa pe al doilea fir să golească containerul, metoda *extrage* fiind şi ea sincronizată. Prin urmare am avea nevoie de ceva asemănător ciclului *while* dar care pe lângă faptul că ţine firul în aşteptare ar trebui să pună temporar obiectul în starea "liber"! Exact acest lucru îl face metoda *wait*: ţine firul în aşteptare şi pune obiectul apelat (la noi, *this*) în starea "liber". Prin urmare nu e nici o problemă pentru al doilea fir să extragă numărul din container. După ce-l extrage, al doilea fir apelează *notifyAll* anunţând primul fir ce e "blocat" în wait-ul din metoda *pune* că poate să-şi reia activitatea, condiţia de golire a containerului fiind satisfăcută. Când are loc pornirea primului fir? Nu se poate spune exact pentru că depinde de maşina virtuală, dar el sigur va porni. Mai mult, maşina virtuală îl va porni în aşa fel încât condiţia de excludere mutuală impusă de clauzele *synchronized* să fie respectate.

Ar mai rămâne un singur aspect de discutat. De ce e nevoie de două perechi de *wait-notifyAll*. Simplu, cazul de deadlock expus de noi presupunea că se umple containerul şi nu se mai poate goli, dar poate apărea şi situaţia simetrică: containerul se goleşte şi nu se mai poate umple. Tratarea situaţiei este exact identică.

Important Aceast exemplu reflectă un caz particular de problemă de tip producător-consumator. Producătorul e firul care pune numere în container, iar consumatorul e firul care ia numere din ACELAŞI container. Generalizarea problemei apare atunci când există mai mulţi producători, mai mulţi consumatori iar în container

se pot pune mai multe numere.

V-aţi întrebat de ce *wait* e cuprins într-un ciclu *while*? Dacă am avea mai multe fire care pun valori în container, atunci se poate întâmpla ca mai multe fire să fie "blocate" în wait-ul din pune. Când al doilea fir extrage un număr, el apelează *notifyAll* deblocând toate firele ce aşteaptă la *wait*-ul asociat. Dar după ce un astfel de fir pune un număr în container, restul de fire trebuie să se întoarcă înapoi în *wait* deoarece condiţia de continuare, deşi a fost pentru scurt timp adevărată, a devenit iarăşi falsă. Prin urmare ciclul *while* se foloseşte pentru a retesta la fiecare deblocare condiţia de continuare. Retestarea condiţiei de contibuare e necesară şi datorită faptului că *notifyAll* deblochează şi eventualele alte fire ce aşteaptă în *wait*-ul din *extrage*. Ele trebuie să-şi verifice condiţia de continuare pentru a vedea dacă apelul la *notifyAll* este sau nu destinat lor.

11.4 Exerciţii

1. Într-o benzinărie există un singur rezervor de benzină de capacitate 1000 de litri. Acest rezervor este alimentat de mai multe cisterne de benzină de capacităţi diferite. Evident, dacă rezervorul nu poate prelua întreaga cantitate de benzină pe care o conţine o cisternă, aceasta va aştepta eliberarea capacităţii necesare şi numai atunci va depune în rezervor întreaga cantitate de benzină conţinută. Golirea cisternelor nu ţine cont de ordinea de sosire a cisternelor în benzinărie.

   De la acest rezervor se alimentează autoturisme, un autoturism preluând cantităţi aleatoare de benzină din rezervor, dar unul preia tot timpul un singur litru (oare de ce?). Evident, dacă rezervorul nu conţine cantitatea cerută de un autoturism, acesta va aştepta până când poate prelua integral întreaga cantitate cerută (care e tot timpul mai mică decât 1000). Alimentarea autoturismelor nu ţine cont de sosirea în benzinărie a autoturismelor.

   Să se implementeze un program care simulează situaţia descrisă mai sus utilizând fire de execuţie. O cisternă va fi modelată printr-un obiect fir de execuţie cu capacitatea specificată prin constructor şi care depune la infinit benzină în rezervorul benzinăriei. Un autoturism va fi modelat tot printr-un obiect fir de execuţie care preia la infinit cantităţi aleatoare de benzină din rezervorul benzinăriei. Se va scrie, de asemenea, o metodă *main* care creează 3 cisterne de capacităţi diferite (mai mici de 1000) şi 5 autoturisme după care declanşează firele de execuţie asociate acestora.
2. Într-un magazin de gresie există doi agenţi de vânzări şi o singură caseriţă. Odată ce un agent face o vânzare el depune într-o coadă de aşteptare unică un bon ce conţine numele agentului şi suprafaţa de gresie vândută. Capacitatea cozii este de 20 de bonuri. Dacă nu mai există loc în coadă agentul aşteaptă eliberarea unui loc pentru a depune bonul. Caseriţa preia bonurile din coadă în ordinea în care au

fost depuse şi emite o chitanţă de forma: Nume Agent - Suprafaţa Vândută - Preţ. Preţul unui metru pătrat de gresie este de 25 de lei. Dacă nu există nici un bon în coada de aşteptare caseriţa aşteaptă.

Să se implementeze problema de mai sus (producător - consumator) folosind trei obiecte fire de execuţie: două pentru simularea agenţilor (producătorii) ce citesc suprafeţele vândute din două fişiere, şi unul pentru simularea caseriţei ce emite chitanţe (consumatorul) scriind "chitanţele" într-un fişier. Implementarea trebuie să garanteze faptul că pentru fiecare bon se emite o chitanţă şi numai una.

3. Cinci filosofi se află aşezaţi în jurul unei mese rotunde. Fiecare are în faţă o farfurie cu mâncare. De asemenea, între oricare doi filosofi există o singură furculiţă. Masa fiind rotundă, rezultă că există cinci furculiţe. Fiecare filosof repetă la infinit următoarea secvenţă de operaţii:
   - gândeşte o perioadă aleatoare de timp, caz în care nu are nevoie de nici o furculiţă (el trebuie să lase furculiţele pe masă când gândeşte).
   - i se face foame, caz în care încearcă să pună stăpânire pe cele două furculiţe din vecinătatea sa.
   - mănâncă o perioadă aleatoare de timp, dar numai după ce a obţinut cele două furculiţe.

   Să se scrie un program Java care simulează activităţile acestor filosofi. Fiecare filosof va avea propriul fir de execuţie care va realiza activităţile descrise mai sus (afişând mesaje corespunzătoare pe ecran). Programul trebuie astfel implementat încât să nu apară situaţii de impas.

   NOTĂ Pentru a realiza cooperarea între filosofi, fiecărei furculiţe trebuie să-i corespundă un obiect. Cel mai bine e să modelaţi furculiţele printr-o clasă *Furculita*. Ce metode trebuie să aibe această clasă? Dacă alegeţi pentru a obţine excluderea mutuală a accesului la o furculiţă blocurile de sincronizare, nu mai e necesară nici o metodă dedicată cooperării. Dacă nu folosiţi blocuri de sincronizare atunci va fi necesară utilizarea şi a cooperării prin condiţii, iar clasa *Furculita* va avea o metodă prin care va trece un obiect furculiţă în starea "furculiţă utilizată" şi o alta prin care va trece un obiect furculiţă în starea "furculiţă neutilizată" (Atenţie: nu există nici o legătură între aceste stări şi starea "ocupat"/"liber" a unui obiect din contextul metodelor sincronizate). Cooperarea prin condiţii e utilizată în ideea că un filosof nu poate trece o furculiţă în starea "utilizată" dacă ea e deja în această stare.

Problema filosofilor este o problemă celebră în contextul programelor concurente. Situaţia de impas poate apare dacă, de exemplu, fiecare filosof a luat furculiţa din dreapta sa şi aşteaptă să preia furculiţa din stânga sa. Într-o astfel de situaţie nici un filosof nu poate continua deoarece

furculiţa din stânga sa este deţinută de filosoful din stânga lui care şi el aşteaptă să preia furculiţa din stânga sa pe care însă nu o poate lua deoarece e deţinută de filosoful din stânga lui ş.a.m.d. Pe lângă situaţia de impas, în această problemă mai poate apare şi o situaţie cunoscută în domeniul programării concurente ca situaţie de *înfometare (starvation)*. Pe scurt, aceasta înseamnă că un filosof nu ajunge niciodată să mănânce, el aşteptând să ia de exemplu furculiţa stângă dar filosoful din stânga sa o lasă jos după care imediat o ia înapoi (şi asta tot timpul). În implementarea voastră nu trebuie să evitaţi şi această situaţie.

Bibliografie

1. James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, *Java Language Specification*, *http://java.sun.com/docs/books/jls*, 2005.