FIRE DE EXECUTARE

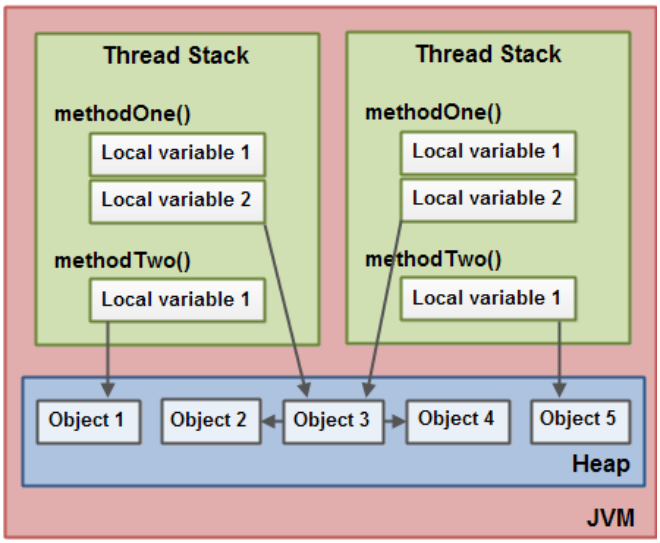
Primele sisteme de operare erau monotasking, respectiv erau capabile să execute un singur program la un moment dat (de exemplu, sistemul de operare MS-DOS). Ulterior, SO au devenit multitasking, respectiv pot rula simultan mai multe programe. Dacă sistemul de calcul este multiprocesor, atunci rularea programelor se realizează efectiv în paralel, iar în cazul sistemelor de calcul monoprocesor rularea lor în paralel este, de fapt, simulată (de exemplu, sistemul de operare Windows).

Un program aflat în executare se numește *proces*. Fiecărui proces i se asociază un segment de cod, un segment de date și resurse (fișiere, memorie alocată dinamic etc.). Procesele NU execută instrucțiuni, ci doar creează un context în care acestea să fie executate. Unitatea de executare a unui proces este *firul de executare*.

Un *fir de executare* este o succesiune secvențială de instrucțiuni care se execută în cadrul unui proces. Orice proces conține cel puțin un fir de executare principal, din care pot fi create alte fire, care, la rândul lor, pot crea alte fire. Un fir de executare nu poate fi rulat independent, ci doar în cadrul unui proces.

Unui fir de executare îi sunt alocate o secvență de instrucțiuni, un set de regiștri și o stivă, proprii acestuia. Firele de executare din cadrul aceluiași proces pot accesa, simultan, resursele procesului părinte (memoria heap și sistemul de fișiere).

De exemplu, să presupunem faptul că avem două fire de executare în cadrul aceluiași program (sursa imaginii: http://tutorials.jenkov.com/java-concurrency/java-memory-model.html):



Fiecare fir de executare are asociat propriul segment de memorie de tip stivă. Variabilele locale declarate în cadrul metodelor sunt salvate în zone diferite de tip stivă, fiecare zonă fiind asociată unui anumit fir, deci aceste zone nu sunt partajate de către cele două fire.

Obiectele sunt alocate în zona de memorie heap, comună tuturor firelor, dar referințele către obiecte sunt stocate în zona de stivă. Atenție, în cazul în care ambele fire vor acționa simultan asupra unui obiect partajat (Object 3), rezultatele obținute pot fi diferite de la o rulare la alta, deoarece ele nu se execută secvențial!

Procesele rulează în contexte diferite, deci comutarea între ele este mai lentă. În schimb, comutarea între firele de executare din cadrul unui proces este mult mai rapidă.

Utilizarea mai multor fire de executare în cadrul unui program va conduce la creșterea performanțelor, mai ales în sistemele multiprocesor sau multicore. Astfel, o operație de lungă durată din cadrul unui program poate fi executată pe un fir separat, în timp ce alte operații se pot executa pe alte fire, în mod independent (nu sunt blocate). De exemplu, într-un browser, o operație de download se execută pe un fir separat, astfel încât să nu blocheze.

Utilizarea firelor de executare implică probleme referitoare la sincronizarea lor, accesarea unor resurse comune (excludere reciprocă), comunicarea între ele etc.

Mașina virtuală Java (JVM) își adaptează strategia de multithreading după tipul sistemului de calcul pe care rulează.

Crearea și pornirea firelor de executare

Clasele și interfețele necesare utilizării firelor de executare în limbajul Java sunt incluse în pachetul `java.lang.Thread`.

Un fir de executare poate fi creat prin două metode:

- extinderea clasei `Thread`
- implementarea interfeței `Runnable`

În ambele variante, trebuie redefinită/implementată metoda `void run()`, scriind în cadrul său secvența de cod pe care dorim să o executăm pe un fir separat.

Exemple:

- prin extinderea clasei `Thread`

```
class FirDeExecutare extends Thread {
    ……
    @Override
    public void run() {
        secvența de cod asociată firului de executare
    }
}
……
FirDeExecutare f = new FirDeExecutare(); // firul nu pornește automat!
f.start(); //trebuie apelată metoda start() care va invoca metoda run()!
```

- prin implementarea interfeței funcționale `Runnable`

```
class FirDeExecutare implements Runnable {
    ……
    @Override
    public void run() {
        secvența de cod asociată firului de executare
    }
}
……
FirDeExecutare f = new FirDeExecutare(); // firul nu pornește automat!
Thread t = new Thread(f);
t.start(); //trebuie apelată metoda start() care va invoca metoda run()!
```

Atenție, pentru lansarea unui fir de executare, indiferent de modalitatea în care acesta a fost creat, trebuie apelată metoda `start()`, care mai întâi va crea contextul necesar unui nou fir de executare (stiva proprie, setul de regiștrii etc.) și apoi va executa metoda `run()` în cadrul noului fir. Dacă am apela direct metoda `run()`, atunci aceasta ar fi executată ca o metodă obișnuită, în cadrul firului curent!

Firele se execută concurent (luptă între ele pentru accesul la resursele comune), motiv pentru care există un arbitru (o componentă a mașinii virtuale Java) numită *planificator* (*thread scheduler*). Acesta gestionează memoria, oferind fiecărui fir spațiul de memorie propriu necesar executării și, în plus, selectează firul care se va executa la un moment dat (devine activ), celelalte fiind trecute în așteptare. Algoritmul de alegere a firului care va deveni activ este dependent de implementarea planificatorului!

Un program se termină când se încheie executarea tuturor firelor lansate din cadrul său.

Exemplu
Clasa `FirDeExecutare` de mai jos utilizează un fir de executare pentru a afișa pe ecran de 100 de ori un caracter `c` primit prin intermediul constructorului clasei:

```
class FirDeExecutare extends Thread
{
    char c;

    public FirDeExecutare(char c)
    {
        this.c = c;
    }

    @Override
    public void run()
    {
        for(int i = 0; i < 100; i++)
            System.out.print(c + " ");
    }
}
```

În clasa `Test_Thread` vom lansa în executare două fire, unul care va afișa cifra 1 și unul care va afișa cifra 2, iar în metoda `main()`, după lansarea celor două fire, vom afișa de 100 de ori cifra 0:

```
public class Test_Thread
{
    public static void main(String[] args)
    {
        FirDeExecutare fir_1 = new FirDeExecutare('1');
        FirDeExecutare fir_2 = new FirDeExecutare('2');

        fir_1.start();
        fir_2.start();

        for(int i = 0; i < 100; i++)
           System.out.print("0 ");
        System.out.println();
    }
}
```

Rulând programul de mai sus de mai multe ori, pe ecran se vor afișa diverse combinații aleatorii formate din cifrele 0, 1 și 2, în care fiecare cifră apare de exact 100 de ori:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

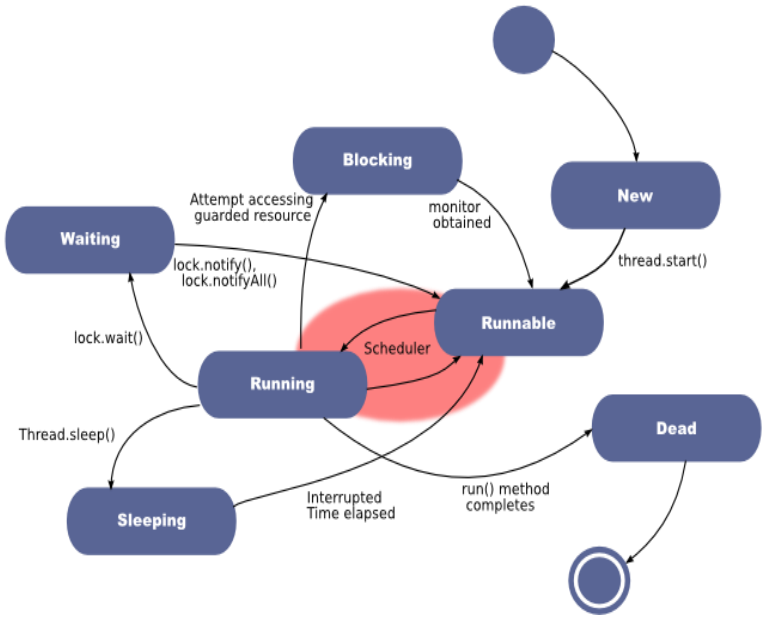
Practic, în acest program sunt executate concurent 3 fire (`fir_1`, `fir_2` și firul principal - cel asociat metodei `main()`), pe care planificatorul le execută într-un mod aleatoriu, astfel: fiecare fir este executat o perioadă de timp, după care el este suspendat și se trece la executarea altui fir, până când toate cele 3 fire își vor încheia executarea. În exemplul de mai sus, se observă faptul că deși firul principal se termină primul (ultimele 4 cifre de 0 sunt afișate pe penultimul rând), executarea programului se încheie doar după ce se termină de executat și celelalte două fire (care afișează cifrele 1 și 2).

Dacă am înlocui instrucțiunile `fir_1.start()` și `fir_2.start()` cu `fir_1.run()` și `fir_2.run()`, pe ecran se va afișa întotdeauna combinația `1 1 ... 1 2 2 ... 2 0 0 ... 0`, fiecare cifră de câte 100 de ori, deoarece metoda `run()` nu va fi executată pe un fir separat, ci va fi executată ca o metodă obișnuită, în firul principal al aplicației!

Ciclul de viață al firelor de executare

Din exemplele anterior prezentate, se observă faptul că modul în care sunt selectate/executate firele de către planificator este aleatoriu, deci este necesară utilizarea unor mecanisme specifice programării concurente pentru managementul acestora.

În figura de mai jos, sunt prezentate stările în care se poate afla un fir de executare (sursa imaginii: http://booxs.biz/EN/java/Threads%20in%20Java.html):



Din figura de mai sus observăm faptul că un fir de executare se poate afla într-una dintre următoarele stări:

- *fir nou* (new) – obiectul de tip `Thread` a fost creat;
- *fir rulabil* (runnable) – după apelarea metodei `start()` a firului de executare, acesta este adăugat în grupul de fire aflate în așteptare (rulabile), deci nu este neapărat executat imediat;
- *fir activ* (running) – firul intră în executare, ca urmare a alegerii sale de către planificator;
- *fir blocat* (waiting/sleeping/blocking) – executarea firului este întreruptă momentan;
- *fir terminat* (dead) – executarea firului s-a încheiat

Există mai multe situații în care dorim ca firul activ să devină inactiv:

- *pentru a da ocazia altor fire să se execute* – se poate utiliza metoda statică `void sleep(long ms)` din clasa `Thread` care suspendă executarea firului curent pentru `ms` milisecunde;
- *pentru a aștepta eliberarea unei resurse partajate* – se poate utiliza metoda `void wait()`, iar firul redevine rulabil după ce un alt fir apelează metoda `void notify()` sau metoda `void notifyAll()` (toate cele 3 metode sunt definite în clasa `Object`!);
- *pentru a aștepta încheierea executării unui alt fir* – se poate utiliza metoda `void join()` care suspendă executarea firului părinte până la terminarea firului curent.

Exemplu:
Vom relua primul exemplu prezentat (în care se afișau pe ecran diverse combinații aleatorii formate din cifrele 0, 1 și 2) și vom modifica doar clasa `Test_Thread`, astfel (liniile de cod adăugate sunt scrise cu font îngroșat):

```
public class Test_Thread
{
    public static void main(String[] args)
    {
        FirDeExecutare fir_1 = new FirDeExecutare('1');
        FirDeExecutare fir_2 = new FirDeExecutare('2');

        fir_1.start();
        fir_2.start();

        try
        {
            fir_1.join();
            fir_2.join();
        }
        catch (InterruptedException ex)
        {
            System.out.println("Eroare fire de executare!");
        }

        for(int i = 0; i < 100; i++)
           System.out.print("0 ");
        System.out.println();
    }
}
```

Cele două apeluri ale metodei `join()` obligă firul părinte (în acest caz, firul principal al aplicației) să aștepte terminarea celor două fire lansate de el în executare (firele **`fir_1`** și **`fir_2`**) înainte să-și continue executarea, deci pe ecran se vor afișa diverse combinații aleatorii formate din cifrele 1 și 2 (de câte 100 de ori fiecare), terminate întotdeauna cu exact 100 de cifre de 0:

```
2 1 1 1 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 2 1 1 2 2 2
2 2 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Un fir de executare poate fi oprit și în mod forțat. Până în versiunea Java 1.5 se putea utiliza metoda `stop()`. Ulterior, din motive de securitate, aceasta a fost considerată ca fiind "depășită", împreună cu alte două metode, respectiv `suspend()` și `resume()`: https://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html.

În prezent, pentru oprirea forțată a unui fir de executare, se utilizează una dintre următoarele metode:

- se folosește o variabilă locală, de obicei de tip boolean, pentru a controla executarea codului din interiorul firului, deci codul din metoda `run()`;

Exemplu
Clasa `FirDeExecutare` de mai jos utilizează un fir de executare pentru a afișa pe ecran numere naturale consecutive în cadrul unei instrucțiuni iterative `while` controlată de variabila booleană `stop`:

```
class FirDeExecutare implements Runnable
{
    int cnt;
    boolean stop;

    public FirDeExecutare()
    {
        cnt = 0;
        stop = false;
    }

    @Override
    public void run()
    {
        while(!stop)
            System.out.println(++cnt + " ");
    }

    public void OprireFir() { stop = true; }
}
```

În clasa `Test_Stop` se va lansa în executare un fir de tipul celui mai sus menționat, iar în firul principal se vor citi șiruri de caractere (atenție, pe ecran se vor afișa numere naturale!) până când utilizatorul va introduce cuvântul "`stop`", după care se va apela metoda `OprireFir()` pentru a întrerupe forțat executarea firului care afișează numerele naturale:

```
public class Test_Stop
{
    public static void main(String[] args)
    {
        String s , aux;

        FirDeExecutare ob = new FirDeExecutare();
        Thread fir = new Thread(ob);
        fir.start();

        Scanner in = new Scanner(System.in);

        aux = "";
        while ((s = in.next()).compareTo("stop") != 0)
            aux = aux + s + " ";

        ob.OprireFir();

        System.out.println("Cuvintele citite: " + aux);
    }
}
```

De exemplu, dacă de la tastatură vom introduce, pe rând, cuvintele "un", "exemplu" și "stop", atunci pe ecran se va afișa un text de tipul următor:

```
1 2 3 4 5 ... 2292929 2292930 2292931 2292932 2292933 2292934 2292935 2292936
2292937 2292938 2292939 Cuvintele citite: un exemplu
```

Observație foarte importantă: Dacă variabila care controlează executarea codului din interiorul metodei `run()` nu este locală, atunci ea trebuie declarată (în exteriorul firului) ca fiind volatilă (de exemplu, `volatile boolean stop`) pentru ca valoarea sa să fie actualizată din memoria principală la fiecare accesare. Altfel, deoarece fiecare fir de executare are propria sa stivă, este posibil să se utilizeze o copie locală a variabilei externe respective, deși, între timp, valoarea variabilei respective a fost modificată de alt fir! Mai multe informații despre acest subiect găsiți în paginile http://tutorials.jenkov.com/java-concurrency/volatile.html și https://dzone.com/articles/java-volatile-keyword-0.

- se folosește metoda `void interrupt()` pentru a întrerupe forțat executarea firului, iar în interiorul metodei `run()` se utilizează metoda statică `boolean interrupted()` din clasa `Thread` pentru a testa dacă firul curent a fost întrerupt sau nu.

Exemplu
Vom relua exemplul anterior folosind metodele menționate mai sus:

```
class FirDeExecutare implements Runnable
{
    int cnt;

    public FirDeExecutare()
    {
        cnt = 0;
    }

    @Override
    public void run()
    {
        while(!Thread.interrupted())
            System.out.println(++cnt + " ");
    }
}


public class Test_Interrupt
{
    public static void main(String[] args)
    {
        String s , aux;

        FirDeExecutare ob = new FirDeExecutare();
        Thread fir = new Thread(ob);
        fir.start();
```

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

            aux = "";
            while ((s = in.next()).compareTo("stop") != 0)
                aux = aux + s + " ";

            fir.interrupt();

            System.out.println("Cuvintele citite: " + aux);
        }
    }
```

Accesarea concurentă a unor resurse comune

Mai devreme, am văzut faptul că mai multe fire pot să partajeze o resursă comună. Un exemplu concret îl reprezintă vânzarea on-line a unor bilete de tren, folosind o aplicație client-server care utilizează o bază de date comună pentru a reține locurile vândute. Evident, există posibilitatea ca, într-un anumit moment, mai mulți operatori să vândă același loc, ceea ce reprezintă o eroare gravă! Evident, în acest caz resursa comună este baza de date, iar operația care poate să conducă la rezervarea de mai multe ori a aceluiași loc este cea de vânzare, deci aceasta este o secțiune critică.

O *secțiune critică* este o secvență de cod care gestionează o resursă comună mai multor de fire de executare care acționează simultan.

Pentru a rezolva o problema de sincronizare de tipul celei precizate anterior, trebuie ca secțiunea critică să se execute prin excludere reciprocă, adică în momentul în care un fir acționează asupra resursei comune, restul firelor vor fi blocate. Astfel, în exemplul de mai sus, în momentul vânzării unui anumit loc de către un operator, toți ceilalți operatori care ar dori să vândă același loc vor fi blocați.

Controlul accesului într-o secțiune critică (la o resursă comună) se face folosind cuvântul cheie `synchronized`.

Exemplu:
Mai întâi, vom considera o clasă `Counter` care implementează un simplu contor:

```
class Counter
{
    private long count;

    Counter() { count = 0; }

    public long getCount() { return count; }

    public void add() { count++; }
}
```

De asemenea, vom considera o clasă `CounterThread` care incrementează de 10000 de ori un contor de tipul `Counter`, utilizând un fir de executare dedicat:

```
class CounterThread extends Thread
{
    private Counter counter = null;

    public CounterThread(Counter counter)
    {
        this.counter = counter;
    }

    @Override
    public void run()
    {
        for (int i = 0; i < 10000; i++)
                counter.add();
    }
}
```

În continuare, vom considera o clasă `CounterThread` care incrementează de 10000 de ori un contor de tipul `Counter`, utilizând un fir de executare dedicat:

```
class CounterThread extends Thread
{
    private Counter counter = null;

    public CounterThread(Counter counter)
    {
        this.counter = counter;
    }

    @Override
    public void run()
    {
        for (int i = 0; i < 10000; i++)
                counter.add();
    }
}
```

În metoda `main()` a clasei `Test_Sincronizare`, mai întâi vom crea un contor `counter` și apoi două fire de executare, `fir_1` și `fir_2`, care vor accesa contorul comun `counter`:

```
public class Sincronizare
{
    public static void main(String[] args) throws InterruptedException
    {
        Counter counter = new Counter();

        Thread thread_1 = new CounterThread(counter);

        Thread thread_2 = new CounterThread(counter);

        thread_1.start();
        thread_2.start();
```

```
        thread_1.join();
        thread_2.join();

        System.out.println("Counter: " + counter.getCount());
    }
}
```

Observați faptul că am apelat metoda `join()` pentru ambele fire de executare, astfel încât în firul principal al aplicației valoarea contorului să fie afișată doar după ce sunt încheiate executările ambelor fire! Rulând programul de mai multe ori, se va afișa, de obicei, o valoare strict mai mică decât valoarea 20000 pe care o anticipam. Acest lucru se întâmplă deoarece, în mai multe momente de timp, ambele fire vor încerca să incrementeze contorul comun (evident, fără a reuși), ceea ce va conduce la o valoare finală eronată a contorului.

Pentru a rezolva această problemă, trebuie să realizăm incrementarea contorului (în metoda `add()` din clasa `Counter`) sub excludere reciprocă, respectiv în momentul în care un fir de executare incrementează contorul, celălalt fir să fie suspendat și abia după ce primul fir termină operația de incrementare a contorului să se reia executarea celui de-al doilea fir.

Pentru asigurarea excluderii reciproce, putem să utilizăm cuvântul cheie `synchronized` în două moduri:

- *la nivel de metodă*, adăugând cuvântul cheie `synchronized` în antetul metodei `add()`:

```
synchronized public void add()
{
      count++;
}
```

- *la nivel de bloc de instrucțiuni*, adăugând cuvântul cheie `synchronized` doar pentru secțiunea critică:

```
public void add()
{
      synchronized(this)
      {
            count++;
      }
}
```

Dacă se apelează o metodă nestatică sincronizată pentru un obiect, atunci alte fire nu mai pot apela, pentru același obiect, nicio altă metodă sincronizată. Totuși, se pot apela alte metode care nu sunt sincronizate!

Dacă se apelează o metodă statică sincronizată pentru un obiect, atunci alte fire nu mai pot apela, pentru nici un alt obiect al clasei respective, nicio altă metodă sincronizată. Totuși, se pot apela alte metode care nu sunt sincronizate. Practic, în acest caz, sincronizarea se realizează la nivel de clasă, ci nu de obiect!

Totuși, utilizarea mai multor metode sincronizate va conduce la un timp de executare mare. Pentru a evita acest lucru, dacă o metodă conține doar un bloc de instrucțiuni care necesită sincronizare (o secțiune critică), atunci se va sincroniza doar blocul respectiv. În acest caz, alte fire pot invoca și alte metode, sincronizate sau nu!

În practică, sunt multe situații în care firele de executare nu trebuie doar să se excludă reciproc, ci să și coopereze. În acest scop, există metode specifice definite în clasa `Object`, respectiv metodele `wait()` și `notify()/notifyAll()`. Pentru a înțelege modalitatea de utilizare a acestor metode, vom considera faptul că asupra unui obiect acționează mai multe fire de executare și unul dintre fire preia controlul exclusiv asupra obiectului, însă nu-și poate finaliza acțiunea fără ca un alt fir să execute o acțiune suplimentară. În acest caz, firul respectiv se va auto-suspenda, folosind metoda `wait()`, trecând astfel într-o stare de așteptare și eliberând controlul asupra obiectului partajat. În acest moment, un alt fir poate prelua controlul exclusiv asupra obiectului pentru a efectua acțiunea suplimentară, iar după efectuarea acesteia, firul respectiv va înștiința firul sau firele aflate în așteptare, folosind metodele `notify()` sau `notifyAll()`, despre faptul că a efectuat o acțiune. Aceste două metode, spre deosebire de metoda `wait()`, nu vor ceda imediat controlul asupra obiectului, respectiv, dacă în codul firului respectiv mai există alte instrucțiuni după apelul unei dintre cele două metode, acestea vor fi executate și abia apoi va fi cedat controlul asupra obiectului partajat.

Problema producător–consumator este un exemplu clasic în care este evidențiată necesitatea de colaborare între două fire de executare: "*Un producător şi un consumator îşi desfăşoară simultan activităţile, folosind în comun o bandă de lungime fixă. Producătorul produce câte un obiect şi îl plasează la un capăt al benzii, iar consumatorul preia câte un obiect de la celălalt capăt al benzii şi îl consumă.*"

În simularea activităților producătorului și consumatorului, trebuie să ținem cont de faptul că producătorul și consumatorul plasează/preiau obiecte pe/de pe banda comună în ritmuri aleatorii, ceea ce poate conduce la următoarele situații limită:

- *producătorul încearcă să plaseze un obiect pe banda plină* – în acest caz producătorul trebuie să se auto-suspende, folosind metoda `wait()`, până în momentul în care consumatorul va prelua cel puțin un obiect de pe banda plină;
- *consumatorul încearcă să preia un obiect de pe banda vidă* – în acest caz consumatorul trebuie să se auto-suspende, folosind metoda `wait()`, până în momentul în care producătorul va plasa cel puțin un obiect pe banda vidă.

După fiecare acțiune reușită de plasare/preluare a unui obiect pe/de pe bandă, producătorul/consumatorul va apela metoda `notify()` pentru a permite deblocarea unui eventual consumator/producător suspendat.

Pentru a implementa această problemă în limbajul Java, vom defini mai întâi clasa `BandaComună`, considerând obiectele ca fiind numere naturale nenule:

```
class BandaComună
{
    private LinkedList<Integer> banda;
    private int dimMaximăBandă;

    public BandaComună(int dimMaximaBanda)
    {
        banda = new LinkedList();
        this.dimMaximăBandă = dimMaximaBanda;
    }
```

```
    public synchronized void PlaseazăObiect(int x) throws InterruptedException
    {
        while (banda.size() == dimMaximăBandă)
            wait();

        banda.add(x);
        System.out.println("Producator: obiect " + x);
        notify();
    }

    public synchronized void PreiaObiect() throws InterruptedException
    {
        while (banda.size() == 0)
            wait();

        int x = banda.remove(0);
        System.out.println("Consumator: obiect " + x);
        notify();
    }
}
```

Observați faptul că operațiile de plasare/preluare a unui obiect pe/de pe banda comună se execută sub excludere reciprocă! De ce a fost nevoie de aceste restricții?

În continuare, definim clasa `Producător`, în cadrul căreia vor fi produse 10 obiecte, identificate prin numerele naturale de la 1 la 10:

```
class Producător extends Thread
{
    private BandaComună banda;

    public Producător(BandaComună banda) { this.banda = banda; }

    @Override
    public void run()
    {
        for (int i = 1; i <= 10; i++)
            try
            {
                Thread.sleep((int) (Math.random() * 100));
                banda.PlaseazăObiect(i);
            }
            catch (InterruptedException e){}
    }
}
```

Pentru a simula mai bine ritmul aleatoriu de plasare a obiectelor pe bandă, am întârziat operația respectivă cu un număr aleatoriu cuprins între 0 și 99 de milisecunde, utilizând metoda `sleep`.

Într-un mod asemănător definim și clasa `Consumator`:

```
class Consumator extends Thread
{
    private BandaComună banda;

    public Consumator(BandaComună banda)
    {
        this.banda = banda;
    }

    @Override
    public void run()
    {
        for (int i = 1; i <= 10; i++)
            try
            {
                Thread.sleep((int) (Math.random() * 100));
                banda.PreiaObiect();
            }
            catch (InterruptedException ex){}
    }
}
```

În clasa `Producător_Consumator` vom simula efectiv activitatea producătorului și consumatorului, utilizând o bandă comună de lungime 5:

```
public class Producător_Consumator
{
    public static void main(String[] args)
    {
        BandaComună b = new BandaComună(5);

        Producător p = new Producător(b);
        Consumator c = new Consumator(b);

        p.start();
        c.start();
    }
}
```

Rulând programul, vom obține diverse variante de simulare a activităților producătorului și consumatorului, una dintre ele fiind următoarea:

```
Producător: obiect 1
Consumator: obiect 1
Producător: obiect 2
Producător: obiect 3
Producător: obiect 4
Consumator: obiect 2
Consumator: obiect 3
Producător: obiect 5
```

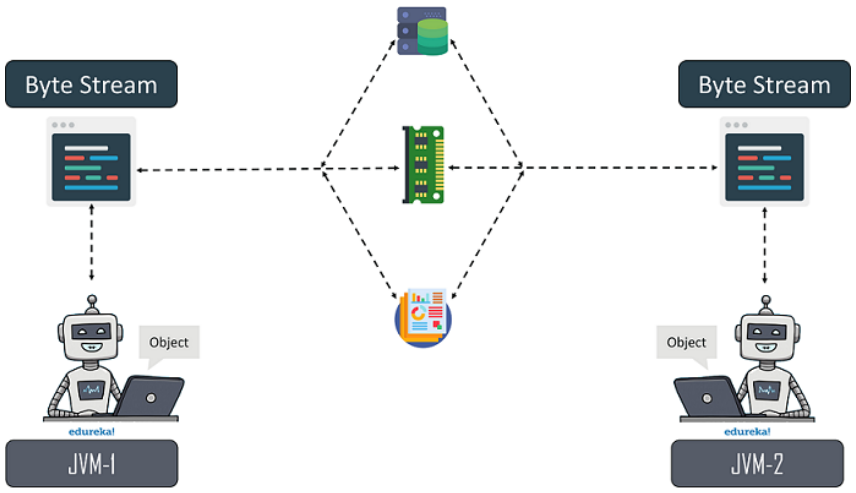
```
Consumator: obiect 4
Producător: obiect 6
Consumator: obiect 5
Producător: obiect 7
Consumator: obiect 6
Producător: obiect 8
Consumator: obiect 7
Producător: obiect 9
Consumator: obiect 8
Producător: obiect 10
Consumator: obiect 9
Consumator: obiect 10
```

Încheiem precizând faptul că programarea folosind fire de executare este un domeniu important și actual al informaticii, tratat pe larg în cadrul unor discipline precum calcul paralel și concurent sau calcul distribuit.

SERIALIZAREA OBIECTELOR

Ciclul de viață al unui obiect este determinat de executarea programului, respectiv obiectele instanțiate în cadrul său sunt stocate în memoria internă, astfel încât, după ce acesta își termină executarea, zona de memorie alocată programului este eliberată. În cadrul aplicațiilor, de cele mai multe ori, se dorește salvarea obiectelor între diferite rulări ale programului sau se dorește ca acestea să fie transmise printr-un un canal de comunicație. O soluție aparent simplă pentru rezolvarea acestei probleme ar constitui-o salvarea stării unui obiect (valorile datelor membre) într-un fișier (text sau binar) și restaurarea ulterioară a acestuia, pe baza valorilor salvate, folosind un constructor al clasei. Totuși, această soluție devine foarte complicată dacă unele date membre sunt referințe către alte obiecte, deoarece ar trebui salvate și restaurate și stările acelor obiecte externe! Mai mult, în acest caz nu s-ar salva funcționalitățile obiectului (metodele sale) și constructorii.

Limbajul Java permite o rezolvare simplă și eficientă a acestei probleme prin intermediul mecanismelor de *serializare* și *deserializare* (sursa imaginii: https://www.edureka.co/blog/serialization-in-java/):



Serializarea este mecanismul prin care un obiect este transformat într-o secvență de octeți din care acesta să poată fi refăcut ulterior, iar *deserializare*a reprezintă mecanismul invers serializării, respectiv dintr-o secvență de octeți serializați se restaurează obiectul original.

Utilizarea mecanismului de serializare prezintă mai multe avantaje:

- obiectele pot fi salvate/restaurate într-un mod unitar pe/de pe diverse medii de stocare (fișiere binare, baze de date etc.);
- obiectele pot fi transmise foarte simplu între mașini virtuale Java diferite, care pot rula pe calculatoare având arhitecturi sau sisteme de operare diferite;
- timpul necesar serializării sau deserializării unui obiect este mult mai mic decât timpul necesar salvării sau restaurării unui obiect pe baza valorilor datelor sale membre (de exemplu, în momentul deserializării unui obiect nu se apelează constructorul clasei respective);
- cea mai simplă și mai rapidă metodă de clonare a unui obiect (*deep copy*) o reprezintă serializare/deserializarea sa într-un/dintr-un tablou de octeți.

Obiectele unei clase sunt serializabile dacă respectiva clasă implementează interfața `Serializable`. Această interfață este una de marcaj, care nu conține nicio metodă abstractă, deci, prin implementarea sa clasa respectivă doar anunță mașina virtuală Java faptul că dorește să-i fie asigurat mecanismul de serializare. O clasă nu este implicit serializabilă, deoarece clasa `java.lang.Object` nu implementează interfața `Serializable`. Totuși, anumite clase standard, cum ar fi clasa `String`, clasele înfășurătoare (wrapper), clasa `Arrays` etc., implementează interfața `Serializable`.

Pentru prezentarea mecanismului de serializare/deseralizare, vom considera definită clasa `Student`, care implementează interfața `Serializable`, având datele membre `String` nume, `int` grupa, un tablou `note` cu elemente de tip `int` pentru a reține notele unui student, `double` medie și o dată membră statică `facultate` de tip `String`, respectiv metodele de tip `set/get` corespunzătoare, metoda `toString()` și constructori:

```
public class Student implements Serializable
{
    private static String facultate;
    private String nume;
    private int grupa, note[];
    private double medie;
    ...
}
```

Serializarea unui obiect se realizează astfel:

- se deschide un flux binar de ieșire utilizând clasa `java.io.ObjectOutputStream`:
  ```
  FileOutputStream file = new FileOutputStream("studenti.bin");
  ObjectOutputStream fout = new ObjectOutputStream(file);
  ```

- se salvează/scrie obiectul în fișier apelând metoda `void writeObject(Object ob)`:
  ```
  Student s = new Student("Ion Popescu", 241, new int[]{10, 9, 10, 7, 8});
  fout.writeObject(s);
  ```

Deserializarea unui obiect se realizează astfel:

- se deschide un flux binar de intrare utilizând clasa `java.io.ObjectInputStream`:
  ```
  FileInputStream file = new FileInputStream("studenti.bin");
  ObjectInputStream fin = new ObjectInputStream(file);
  ```

- se citește/restaurează obiectul din fișier apelând metoda `Object readObject()`:
  `Student s = (Student)fin.readObject();`

Mecanismul de serializare a unui obiect presupune salvarea, în format binar, a următoarelor informații despre acesta:
- denumirea clasei de apartenență;
- versiunea clasei de apartenență, implicit aceasta fiind hash-code-ul acesteia, calculat de către mașina virtuală Java;
- valorile datelor membre de instanță.
- antetele metodelor membre.

Observații:

- Implicit NU se serializează datele membre statice și nici corpurile metodelor, ci doar antetele lor.
- Explicit NU se serializează datele membre marcate prin modificatorul transient (de exemplu, s-ar putea să nu dorim salvarea anumitor informații confidențiale: salariul unei persoane, parola unui utilizator etc.).
- Serializarea nu tine cont de specificatorii de acces, deci se vor serializa și datele/metodele private!
- În momentul sterilizării unui obiect se va serializa întregul graf de dependențe asociat obiectului respectiv, adică obiectul respectiv și toate obiectele referite direct sau indirect de el.

Exemplu:
Considerăm clasa `Nod` care modelează un nod al unei liste simplu înlănțuite:

```
class Nod implements Serializable
{
    Object data;
    Nod next;

    public Nod(Object data)
    {
        this.data = data;
        this.next = null;
    }
}
```

Folosind clasa `Nod`, vom construi o listă circulară, formată din numerele naturale cuprinse între 1 și 10, pe care apoi o vom salva/serializa în fișierul binar `lista.ser`, scriind în fișier doar primul său nod (obiectul `prim`) – restul nodurilor listei vor fi salvate/serializate automat, deoarece ele formează graful de dependențe asociat obiectului `prim`:

```
public class Serializare_listă_circulară
{
    public static void main(String[] args)
    {
        Nod prim = null, ultim = null;

        for (int i = 1; i <= 10; i++)
        {
            Nod aux = new Nod(i);
```

```
            if (prim == null) prim = ultim = aux;
            else
            {
                ultim.next = aux;
                ultim = aux;
            }
        }
        ultim.next = prim;

        System.out.println("Lista care va fi serializată:");
        Nod aux = prim;
        do
        {
            System.out.print(aux.data + " ");
            aux = aux.next;
        }
        while(aux != prim);

        try (ObjectOutputStream fout = new ObjectOutputStream(
                                        new FileOutputStream("lista.ser")))
        {
            fout.writeObject(prim);
        }
        catch (IOException ex) { System.out.println("Excepţie: " + ex); }
    }
}
```

Pentru a restaura lista circulară salvată/serializată în fișierul binar `lista.ser`, vom citi/deserializa din fișier doar primul său nod (obiectul `prim`), iar restul nodurilor listei vor fi restaurate/deserializate automat, deoarece ele formează graful de dependențe asociat obiectului `prim` (evident, clasa `Nod` trebuie să fie vizibilă):

```
public class Deserializare_listă_ciculară
{
    public static void main(String[] args)
    {
        try (ObjectInputStream fin = new ObjectInputStream(
                                        new FileInputStream("lista.ser")))

        {
            Nod prim = (Nod) fin.readObject();



            System.out.println("Lista deserializata:");
            Nod aux = prim;
            do
            {
                System.out.print(aux.data + " ");
                aux = aux.next;
            }
            while(aux != prim);
```

```
            System.out.println();
        }
        catch (Exception ex)
        {
            System.out.println("Excepție: " + ex);
        }
    }
}
```

În exemplul prezentat, graful de dependențe asociat obiectului `prim` este unul ciclic, deoarece lista este circulară (deci există o referință indirectă a obiectului `prim` către el însuși), dar mecanismul de serializare gestionează fără probleme o astfel de situație complicată!

- Dacă un obiect care trebuie serializat conține referințe către obiecte neserializabile, atunci va fi generată o excepție de tipul `NotSerializableException`.
- Dacă o clasă serializabilă extinde o clasă neserializabilă, atunci datele membre accesibile ale superclasei nu vor fi serializate. În acest caz, superclasa trebuie să conțină un constructor fără argumente pentru a inițializa în procesul de restaurare a obiectului datele membre moștenite.
- Dacă se modifică structura clasei aflată pe mașina virtuală care realizează serializarea obiectelor (de exemplu, prin adăugarea unui câmp nou privat, care, oricum, nu va fi accesibil), fără a se realiza aceeași modificare și pe mașina virtuală destinație, atunci procesul de deserializare va lansa la executare excepția `InvalidClassException`. Excepția apare deoarece cele două clase nu mai au aceeași versiune. Practic, versiunea unei clase se calculează în mod implicit de către mașina virtuală Java printr-un algoritm de hash care este foarte sensibil la orice modificare a clasei. În practică, sunt diferite situații în care se dorește modificarea clasei pe mașina virtuală care realizează procesul de serializare (de exemplu, adăugând date sau metode private care vor fi utilizate doar intern), fără a afecta, însă, procesul de deserializare. O soluție în acest sens o constituie introducerea unei noi date membre în clasă, `private static final long serialVersionUID`, prin care se definește explicit versiunea clasei - caz în care mașina virtuală Java nu va mai calcula, implicit, versiunea clasei respective pe baza structurii sale. Un exemplu bun în acest sens se găsește în pagina https://www.baeldung.com/java-serial-version-uid.

EXTERNALIZAREA OBIECTELOR

Deși mecanismul de serializare este unul foarte puternic și util, totuși, el are și câteva dezavantaje:

- serializarea unui obiect nu ține cont de modificatorii de acces ai datelor membre, deci vor fi salvate în format binar și datele de tip `protected/private`, ceea ce permite reconstituirea valorilor lor prin tehnici de *reverse engineering* (https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/Chapters/3_5-Serialization.pdf);
- serializarea nu salvează datele membre statice;
- serializarea unui obiect necesită destul de mult spațiu de stocare, deoarece se salvează multe informații auxiliare (de exemplu, cele referitoare la superclasele clasei corespunzătoare obiectului);
- serializarea unui obiect se realizează destul de lent, fiind un proces recursiv vizavi de superclasă și/sau referințe către alte obiecte.

Dezavantajele menționate anterior pot fi ameliorate sau eliminate folosind mecanismul de *externalizare,* care este, de fapt, o serializare complet controlată de către programator (implicit se salvează doar numele clasei). Astfel, programatorul poate decide datele care vor fi salvate în format binar, precum și modalitatea utilizată pentru salvarea lor. De exemplu, pentru a asigura confidențialitatea unei date membre a unui obiect la serializare/deserializare (de exemplu, salariul unui angajat), este necesară criptarea/decriptarea întregului obiect, ceea ce va crește considerabil durata celor două procese. Folosind mecanismul de externalizare, se poate cripta/decripta doar data membră respectivă!

Externalizarea obiectelor unei clase este condiționată de implementarea interfeței `Externalizable`:

```
public interface Externalizable extends Serializable
{
      public void writeExternal(ObjectOutput out) throws IOException;
      public void readExternal(ObjectInput in) throws IOException,
                                                      ClassNotFoundException;
}
```

Practic, în cadrul celor două metode `writeExternal` și `readExternal`, programatorul își poate implementa proprii algoritmi de salvare și restaurare a unui obiect.

Exemplu:
Vom prezenta modul în care mecanismul de externalizare poate fi aplicat în cazul clasei `Student`, menționată anterior:

```
public class Student implements Externalizable
{
    public static String facultate;

    String nume;
    int grupa, note[];
    double medie;
    ..............................


    //se va salva denumirea facultății (prin serializare nu ar fi fost salvată,
    //deoarece este o dată membră statică)
    //nu se vor salva notele unui student
    //media va fi "criptată" folosind formula 2*medie+3
    @Override
    public void writeExternal(ObjectOutput out) throws IOException
    {
           out.writeUTF(facultate);
           out.writeUTF(nume);
           out.writeInt(grupa);
           out.writeDouble(2*medie+3);
           out.writeObject(note);
    }
```

```
    //media va fi "decriptată" folosind formula inversă (medie-3)/2
    @Override
    public void readExternal(ObjectInput in) throws IOException,
                                                    ClassNotFoundException
    {
           facultate = in.readUTF();
           nume = in.readUTF();
           grupa = in.readInt();
           medie = in.readDouble();
           medie = (medie - 3)/2;
    }
}
```

După implementarea celor două metode `writeExternal` și `readExternal`, salvarea/restaurarea datelor se va realiza la fel ca și în cazul serializării, apelând metodele `writeObject` și `readObject`.

Exemplu:
Considerăm un tablou `s` cu elemente de tip `Student`:

```
Student s[] = new Student[5];
Student.facultate =  "Facultatea de Matematica si Informatica";

s[0] = new Student(...);
.......................
```

Salvarea tabloului `s` într-un fișier binar `studenti_extern.ser` se poate realiza astfel:

```
try(ObjectOutputStream fout = new ObjectOutputStream(
                                  new FileOutputStream("studenti_extern.ser")))
{
     fout.writeObject(s);
}
catch (IOException ex)
{
     System.out.println("Excepţie: " + ex);
}
```

Restaurarea tabloului `s` din fișierul binar `studenti_extern.ser` se poate realiza astfel:

```
Student s[];

try(ObjectInputStream fin = new ObjectInputStream(
                                  new FileInputStream("studenti_extern.ser")))
{
     s = (Student [])fin.readObject();
}
catch (Exception ex)
{
     System.out.println(ex);
}
```

Un aspect pe care nu trebuie să-l pierdem din vedere în momentul utilizării externalizării este faptul că se vor pierde toate facilitățile puse la dispoziție de mecanismul de serializare (salvarea automată a informațiilor despre superclasele clasei respective, salvarea automată a grafului de dependențe etc.), deci un programator va trebui să le implementeze explicit!