Curs 8

Bazele multithreading

Există doua tipuri distincte de multitasking: unul bazat pe procese şi al doilea bazat pe fire de execuție sau thread-uri. Este important să facem distincția dintre cele două. Un proces este, în esență, un program ce se află în execuție. De aceea multitasking bazat pe procese înseamnă faptul că două sau mai multe programe rulează în acelaşi timp. De exemplu, compilatorul Java va permite crearea unor noi programe, în timp ce folosiți un editor de text, sau navigați pe Internet.

În cazul multitasking bazat pe thread-uri, acesta este cea mai mică unitate ce va fi programată de dispecer. Aceasta înseamnă că un program poate realiza mai multe lucruri simultan. De exemplu aplicația de mail Outlook, sau un client de mail va permite si trimiterea/primirea de mail-uri in timp ce compuneți un nou mail, sau verificați lista de priorități dintr-o zi, sau căutați un anumit mail, etc.

Avantajul principal al multithreading este că permite scrierea unor programe foarte eficiente, deoarece se elimină spațiile inutile temporale prezente în multe programe, atunci când acesta este *idle*. Majoritatea dispozitivelor I/O sunt mai lente decât CPU. De aceea programul va petrece majoritatea execuției aşteptând informația de la aceste dispozitive. În acest timp, în cazul folosirii multithreading, se pot executa instrucțiuni care să realizeze alte sarcini. De exemplu, în timp ce un program trimite un fişier pe Internet, altă parte a programului poate citi de la tastatura, următoarea bucată ce va fi trimisă.

Un fir de execuție este o parte dintr-un proces executat secvențial, o serie de instrucțiuni ce vor fi executate secvențial. Folosirea a două sau mai multe thread-uri duce la execuția în paralel a acestor serii de instrucțiuni în cadrul unui proces. Mai jos avem reprezentat modul de funcționare al unui proces pe un singur thread.

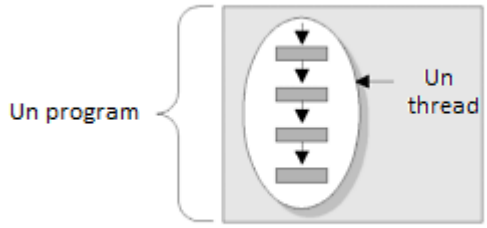

Figura 1. Un proces cu un singur fir de execuție

Adevărata utilitate a unui thread intervine în cazul în care mai multe fire de execuție coexistă în acelaşi proces.

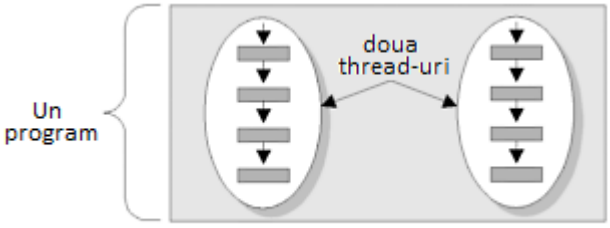

Figura 2. Un proces cu două fire de execuție

Un fir de execuție se poate afla în mai multe stări. Poate fi *running*, adică în execuție. Poate fi *ready to run*, în clipa în care dispecerul îi va asigura timpul în CPU. Un thread aflat în execuție poate fi suspendat, ceea ce înseamnă că este oprit pentru o perioadă. Poate fi reluat, mai târziu, adică va intra în starea de *resumed*. Un thread poate fi *blocat* în momentul când aşteaptă o resursă. Un thread poate fi terminat atunci când execuția se încheie şi nu va mai fi reluat.

Figura de mai jos reprezintă stările unui fir de execuție.

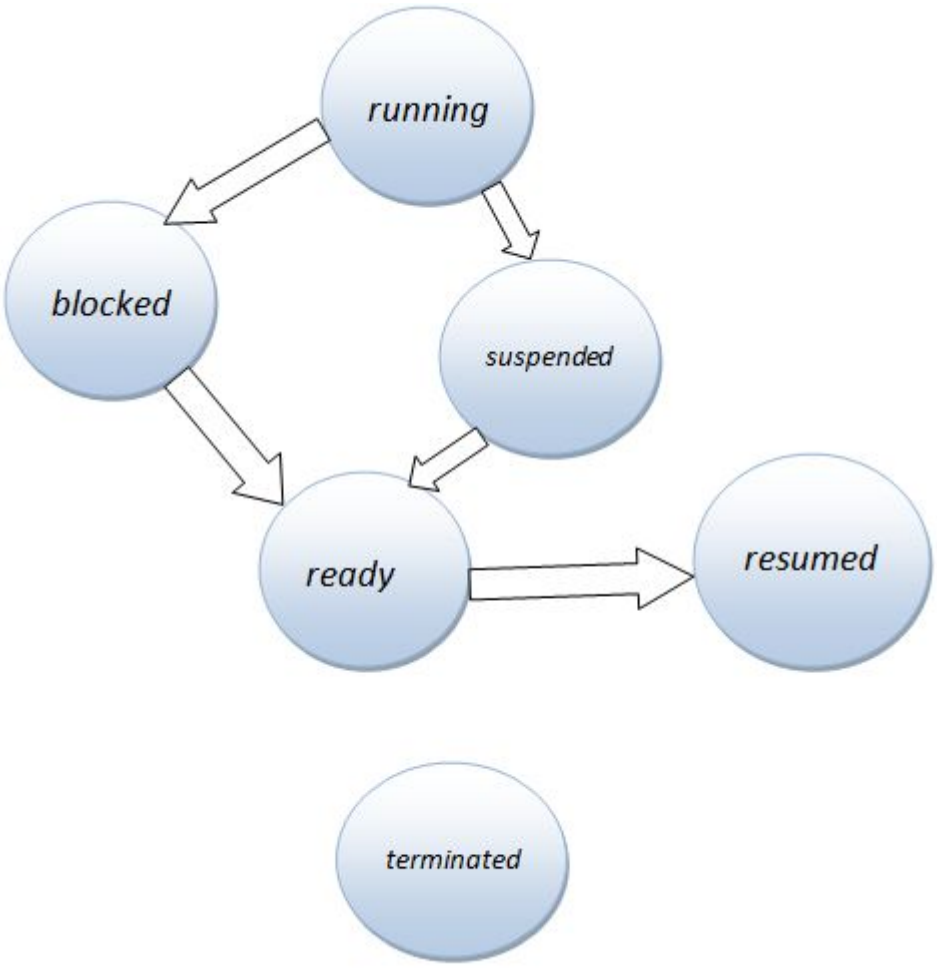

Figura 3. Stările de execuție ale unui thread

Pe lângă aceste caracteristici, mai este nevoie de o anumită funcționalitate, numită *sincronizare*, ce permite execuția firelor într-o manieră controlată.

Față de modul în care limbajul C tratează această problemă a multithreading-ului , Java ascunde unele detalii , tocmai pentru a înlesni şi a face cât mai convenient lucrul cu thread-uri.

Clasa Thread şi interfaţa Runnable

În Java, sistemul multithreading este construit pe clasa *Thread* şi interfaţa *Runnable*. Pentru a crea un nou fir de execuţie, va trebui să extindem clasa Thread sau să implementăm interfaţa Runnable.

Clasa *Thread* defineşte câteva metode care ajută la tratarea situaţiilor ce intervin în lucrul cu thread-uri. Iată cele mai folosite:

Medoda	**Descriere**
final String getName( )	Se obţine numele thread-ului
final int getPriority( )	Se obţine prioritatea thread-ului
final boolean isAlive( )	Determină faptul că un thread mai este în execuţie sau nu.
final void join( )	Aşteaptă terminarea unui thread pentru a continua.
void run( )	Entry point pentru un thread.
static void sleep(long milliseconds)	Suspendă un thread un număr de milisecunde
void start( )	Începe execuţia unui fir prin apel *run()*

Toate procesele au cel puţin un fir de execuţie ce se numeşte firul principal sau *main thread*. Acesta se va executa înainte celorlalte eventual fire de execuţie ce vor fi create ulterior.

Crearea unui fir de execuţie

Un fir de execuţie se poate crea prin instanţierea unui obiect de tip *Thread*. Clasa *Thread* încapsulează un obiect ce implementează interfaţa *Runnable*. Există două moduri de a crea un fir:

1. Implementarea interfeţei *Runnable*
2. Extinderea (moştenirea) clasei *Thread*

Interfaţa Runnable

Această interfaţă defineşte doar o metodă numită **run()** declarată astfel:

```
public void run()
```

În cadrul acestei metode, veţi defini codul ce constituie noul fir de execuţie. Metoda *run()* poate apela alte metode, folosi alte clase, şi declara variabile ca orice metodă. Diferenţa majoră este că *run()* reprezintă *entry point* pentru un nou thread din cadrul procesului. Acest thread îşi încheie execuţia atunci când se iese din funcţia run (cu *return*).

Modul de lucru este următorul:

1. Se creează o clasă ce implementează interfaţa Runnable.
2. Se instanţiază un obiect de tip Thread, într-una din metodele din acea clasă (chiar si in constructorul clasei). Constructorul unui *Thread* va fi:

```
Thread(Runnable threadOb)
```

3. Odată creat, noul obiect de tip Thread, nu va porni până când nu apelăm metoda *start()*. În principiu apelul metodei *start()* înseamnă apelul metodei *run().* Apelul metodei start este:

```
void start()
```

Mai jos avem un exemplu de folosirea a unui nou fir de execute:

```
// crearea unui fir de execute
//prin implementarea interfeței Runnable
class MyThread implements Runnable
{
    int count;
    String threadName;
    MyThread(String name)
    {
        count = 0;
        threadName = name;
    }
    // Entry point al firului
    public void run()
    {
        System.out.println(threadName + " starting.");
        try
        {
            do
            {
                //suspend thread-ul curent
                Thread.sleep(500);
                System.out.println("In " + threadName +
                ", count is " + count);
                count++;
            } while(count < 10);
        }
        catch(InterruptedException exc)
        {
              System.out.println(threadName + " interrupted.");
        }
        System.out.println(threadName + " terminating.");
    }
}
class DemoThread
{
    public static void main(String args[])
    {
        System.out.println("Main thread starting.");
        // mai întâi se construieşte obiectul
        // ce va conţine thread-ul
        MyThread mt = new MyThread("Child #1");
        // se construieşte un fir pe baza obiectului
        Thread newThrd = new Thread(mt);
        // incepe noul fir de execute
        newThrd.start();
        do
```

```
            {
                System.out.print(".");
                try
                {
                    Thread.sleep(100);
                }
                catch(InterruptedException exc)
                {
                    System.out.println("Main thread interrupted.");
                }
            } while (mt.count != 10);
            System.out.println("Main thread ending.");
        }
    }
```

Prima dată, clasa `MyThread` implementează *Runnable*. Aceasta înseamnă că un obiect de tipul `MyThread` va fi folosit pentru a crea un thread şi deci, va fi parametrul unui constructor *Thread*.

În interiorul metodei *run()*, există o buclă *while*, ce contorizează de la 0 la 9. Metoda `Thread.sleep(500);` are ca scop, suspendarea firului de execuție curent timp de 500 de milisecunde.

În clasa `DemoThread`, în metoda *main()*, se creează un nou obiect, de tip `MyThread` care va fi apelat ulterior de `newThrd`:

```
MyThread mt = new MyThread("Child #1");
Thread newThrd = new Thread(mt);
// incepe noul fir de execute
newThrd.start();
```

Obiectul `mt` este folosit pentru a crea un *Thread*, iar acest lucru este posibil pentru că `MyThread` implementează *Runnable* . Execuția unui thread începe cu `start`. Pe firul principal de execuție, instrucțiunile vor fi rulate ca şi când *start()* ar fi o metodă obişnuită. Firul principal de execuție va rula o buclă while în care aşteaptă ca `mt.count` să ajungă la 10. Acest lucru se va întâmpla, datorită faptului că în thread-ul secundar *count* creşte.

Iată rezultatul rulării acestui program:

```
Main thread starting.
.Child #1 starting.
....In Child #1, count is 0
.....In Child #1, count is 1
.....In Child #1, count is 2
.....In Child #1, count is 3
.....In Child #1, count is 4
.....In Child #1, count is 5
.....In Child #1, count is 6
.....In Child #1, count is 7
.....In Child #1, count is 8
.....In Child #1, count is 9
Child #1 terminating.
Main thread ending.
```

O diagrama a acestui program ar fi următoarea:

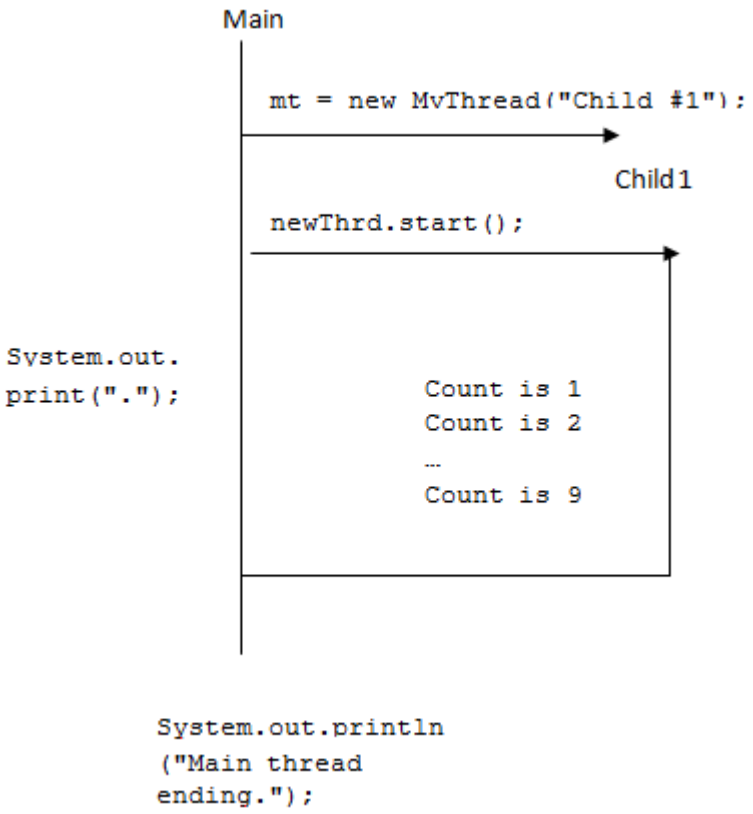

Figura 4. Diagrama execuției unui fir

Îmbunătățiri aduse exemplului

Programul anterior este valid, însă poate fi optimizat:

1. Putem avea un thread care să înceapă execuția la instanțierea obiectului. În cazul `MyThread` aceasta se face instanțiind un obiect de tip `Thread` în constructorul lui `MyThread`.
2. Nu mai este nevoie ca `MyThread` să aibă un obiect de tip `Thread` cu un nume, deoarece modificăm numele la instanțierea noului `Thread`. Pentru aceasta avem următorul constructor:

   ```
   Thread(Runnable threadOb, String name)
   ```

   Se poate obține ulterior, numele *Thread*-ului prin apelul metodei *getName*:

   ```
   final String getName( )
   ```

   Setarea numelui unui thread, după ce acesta a fost creat, se face cu *setName()*:

   ```
   final void setName(String threadName)
   ```

Iată mai jos o versiune a programului anterior, cu îmbunătățirile specificate:

```
// crearea unui fir de execute
//prin implementarea interfeței Runnable
class MyThread implements Runnable
{
    int count;
    //firul de execute cu care vom lucra
    Thread thread;
    MyThread(String name)
    {
        count = 0;
        //instanțiem un nor fir
        thread = new Thread(this, name);
        //firul va porni la apelul constructorului
        //adică la instanțierea lui MyThread
        thread.start();
    }
    // Entry point al firului
    public void run()
    {
        System.out.println(thread.getName() + " starting.");
        try
        {
            do
            {
                //suspend thread-ul curent
                Thread.sleep(500);
                System.out.println("In " + thread.getName() +
                ", count is " + count);
                count++;
            } while(count < 10);
        }
        catch(InterruptedException exc)
        {
              System.out.println(thread.getName() + " interrupted.");
        }
        System.out.println(thread.getName() + " terminating.");
    }
}
class DemoThread
{
    public static void main(String args[])
    {
        System.out.println("Main thread starting.");
        //acum thread-ul nou va porni automat
        //la instanțierea lui mt
        MyThread mt = new MyThread("Child #1");
```

```
        do
        {
            System.out.print(".");
            try
            {
                Thread.sleep(100);
            }
            catch(InterruptedException exc)
            {
                System.out.println("Main thread interrupted.");
            }
        } while (mt.count != 10);
        System.out.println("Main thread ending.");
    }
}
```

Modificările majore sunt: mutarea obiectului de tip `Thread` în cadrul clasei `MyThread` şi pornirea acestui fir din constructorul clasei `MyThread`.

Moştenirea unui Thread

Implementarea interfeței Runnable este un mod de a crea o clasă ce poate instanția fire. A doua ar fi extinderea unei clase numită *Thread*. Atunci când o clasă extinde *Thread*, va trebui să suprascrie metoda *run()*, care este un entry point pentru noul fir. De asemenea trebuie să apeleze *start()* pentru ca noul fir să îşi înceapă execuția.

Pornind de la exemplul anterior, putem transforma acesta aplicând moştenirea lui *Thread*:

1. Se schimba declarația lui *MyThread* ca să extindă *Thread*:

```
class MyThread extends Thread
{
….
```

2. Se şterge această declarație, nu mai avem nevoie de ea:

```
Thread thread;
```

Variabila `thread` nu mai are nici o utilitate, din moment ce *MyThread* este un *Thread*.

3. Se schimbă constructorul lui MyThread ca să arate astfel:

```
// construirea unui fir de execuție.
MyThread(String name)
{
```

```
        super(name); // apelez constructor cu nume
        count = 0;
        start(); // încep execuția
    }
```

Apelul lui super(name); este în fapt, apelul unui *Thread* cu parametru un String:

```
Thread(String name);
```

4. Se schimbă *run()* astfel încât acum apelează *getName().*
   După aceste modificări programul arată astfel:

```
class MyThread extends Thread
{
    int count;
    //firul de execuție cu care vom lucra
    Thread thread;
    MyThread(String name)
    {
       super(name);
       count = 0;
       this.start();
    }
    // Entry point al firului
    public void run()
    {
        System.out.println(getName() + " starting.");
        try
        {
            do
            {
                //suspend thread-ul curent
                Thread.sleep(500);
                System.out.println("In " + getName() +
                ", count is " + count);
                count++;
            } while(count < 10);
        }
        catch(InterruptedException exc)
        {
              System.out.println(getName() + " interrupted.");
        }
        System.out.println(getName() + " terminating.");
    }
}
```

Clasa `DemoThread` nu suferă nici un fel de modificări față de programul precedent.

Crearea mai multor fire de execuție

Exemplele anterioare conțineau doar un singur fir de execuție. Totuşi, programul poate crea oricâte astfel de fire. Următorul program creează trei thread-uri:

```
class MyThread implements Runnable
{
    int count;
    Thread thread;
    // Constructorul ce creează un nou thread
    MyThread(String name)
    {
        thread = new Thread(this, name);
        count = 0;
        thread.start(); // incepe execuția
    }
    //entry point in firul de execuție
    public void run()
    {
        System.out.println(thread.getName() + " starting.");
        try
        {
            // se măreşte contorul
            do
            {
                Thread.sleep(500);
                System.out.println("In " + thread.getName() +
                ", count is " + count);
                count++;
            } while(count < 10);
        }
        catch(InterruptedException exc)
        {
            System.out.println(thread.getName() + " interrupted.");
        }
        System.out.println(thread.getName() + " terminating.");
    }
}
class ThreeChildThreads {
    public static void main(String args[])
    {
        System.out.println("Main thread starting.");
        //cream trei fire de execuție
        MyThread mt1 = new MyThread("Child #1");
        MyThread mt2 = new MyThread("Child #2");
        MyThread mt3 = new MyThread("Child #3");
        //care vor fi automat lansate
        //in bucla while, thread-ul principal
```

```
        //aşteaptă ca toate firele sa se încheie
        do
        {
          System.out.print(".");
          try
          {
             Thread.sleep(100);

          }
          catch(InterruptedException exc)
          {
              System.out.println("Main thread interrupted.");
          }
        } while (mt1.count < 10 || mt2.count < 10 || mt3.count < 10);
        System.out.println("Main thread ending.");
    }
}
```

Cele trei fire de execuție vor rula în paralel cu firul principal, acesta aşteptând ca cele trei sa îşi încheie execuția. O diagramă a acestei funcționalități este în figura de mai jos:

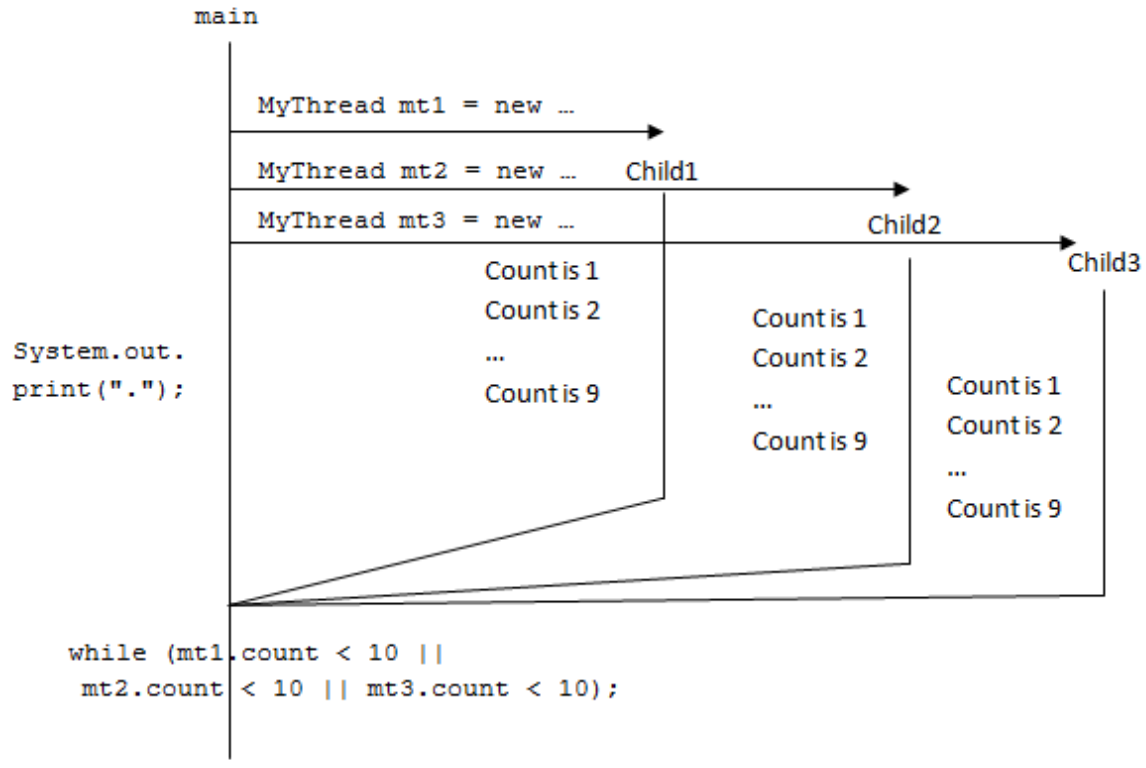

Figura 5. Diagrama execuției pe mai multe fire

Rezultatul rulării acestui program este:

```
Main thread starting.
Child #1 starting.
.Child #2 starting.
Child #3 starting.
....In Child #1, count is 0
In Child #2, count is 0
In Child #3, count is 0
.....In Child #1, count is 1
In Child #3, count is 1
In Child #2, count is 1
.....In Child #1, count is 2
In Child #2, count is 2
In Child #3, count is 2
.....In Child #1, count is 3
In Child #2, count is 3
In Child #3, count is 3
.....In Child #1, count is 4
In Child #2, count is 4
In Child #3, count is 4
.....In Child #1, count is 5
In Child #2, count is 5
In Child #3, count is 5
.....In Child #1, count is 6
In Child #2, count is 6
In Child #3, count is 6
.....In Child #1, count is 7
In Child #3, count is 7
In Child #2, count is 7
.....In Child #1, count is 8
In Child #3, count is 8
In Child #2, count is 8
.....In Child #1, count is 9
Child #1 terminating.
In Child #3, count is 9
Child #3 terminating.
In Child #2, count is 9
Child #2 terminating.
Main thread ending.
```

Acest rezultat depinde de sistemul pe care este rulat şi poate diferi.

Întrebarea firească este de ce avem nevoie de două moduri de a crea fire de execuție (unul prin extinderea Thread şi altul prin implementarea Runnable) şi care este mai performant?

Răspunsul este că o clasă *Thread* defineşte o serie de metode ce pot fi suprascrise de clasa ce derivă din *Thread*. Singura care se impune a fi suprascrisă este *run()*. Aceasta este şi condiția atunci când clasa implementează interfața *Runnable*. Aşa că dacă nu se doreşte suprascrierea altor metode din *Thread*, atunci este mai bine să folosim a doua metodă, pentru simplitate.

Când se încheie un fir de execuție?

Este util să ştim când un fir de execuție s-a încheiat, pentru a controla logica şi fluxul programului. În exemplele anterioare, acest lucru a fost posibil datorită unei variabile de control şi anume *count*. Aceasta este o soluție slabă din punct de vedere tehnic. Clasa *Thread* oferă două moduri de a determina faptul că un fir de execuție s-a încheiat. Primul este prin funcția *isAllive()*:

```
final boolean isAllive()
```

Această metodă returnează *true*, dacă firul de execuție, pentru care a fost apelată, încă rulează, şi *false* altfel. Pentru a verifica funcționalitatea *isAlive()*, vom modifica versiunea clasei `ThreeChildThreads`.

```
class ThreeChildThreads {
    public static void main(String args[])
    {
        System.out.println("Main thread starting.");
        //cream trei fire de execuție
        MyThread mt1 = new MyThread("Child #1");
        MyThread mt2 = new MyThread("Child #2");
        MyThread mt3 = new MyThread("Child #3");
        //care vor fi automat lansate
        //in bucla while, thread-ul principal
        //aşteaptă ca toate firele sa se încheie
        do
        {
          System.out.print(".");
          try
          {
             Thread.sleep(100);

          }
          catch(InterruptedException exc)
          {
              System.out.println("Main thread interrupted.");
          }
        } while (mt1.thread.isAlive() || mt2.thread.isAlive() ||
mt3.thread.isAlive() );
        System.out.println("Main thread ending.");
    }
}
```

După cum se poate observa, acest program este la fel ca şi cel anterior, cu o modificare: condiția din *while* se bazează pe această funcție, oferind robustețe aplicației.

Cel de-al doilea mod de a aştepta ca un fir să îşi încheie execuția este *join()*:

```
final void join() throws IntrerruptedException
```

Această metodă va impune aşteptarea terminarea firului pentru care a fost apelată. Numele provine de la un concept de crearea a unui thread, lansarea a acestuia şi aşteptarea terminării lui pentru ca cel care a creat firul să se “unească” cu acesta. Există forme adiționale ale acestei funcții, prin care se permite specificarea unui număr de milisecunde, sau nanosecunde, timp de aşteptare ca acel fir să se încheie. Mai jos este exemplul pentru această funcție:

```
class ThreeChildThreads
{
    public static void main(String args[])
    {
        System.out.println("Main thread starting.");
        //cream trei fire de execuție
        MyThread mt1 = new MyThread("Child #1");
        MyThread mt2 = new MyThread("Child #2");
        MyThread mt3 = new MyThread("Child #3");
        //care vor fi automat lansate
        //in bucla while, thread-ul principal
        //aşteaptă ca toate firele sa se încheie

          try
          {
            mt1.thread.join();
            System.out.println("Child 1 joined.");
            mt2.thread.join();
            System.out.println("Child 2 joined.");
            mt3.thread.join();
            System.out.println("Child 3 joined.");
          }
          catch(InterruptedException exc)
          {
              System.out.println("Main thread interrupted.");
          }
        System.out.println("Main thread ending.");
    }
}
```

Aşteptarea ca un fir de execuție să se încheie are loc prin apelurile de tipul:

```
mt1.thread.join();
```

Priorități în fire de execuție

Fiecare fir de execuție are asociat o anumită prioritate. Aceasta este determinată de cât timp în CPU este alocat acelui thread. În general prioritățile joase denotă puțin timp, iar cele mari denotă mai mult timp în CPU. Evident, cât de mult timp va avea acces la CPU, un fir de execuție, influențează direct, fluxul logic al programului.

Este important să înțelegem factorii care influențează accesul la CPU. De exemplu, dacă un fir cu prioritate mare aşteaptă o resursă auxiliară, atunci el va fi blocat, şi alt fir cu o prioritate mai mică va avea acces. Totuşi odată ce thread-ul cu prioritatea mai mare are resursa, va opri execuția firului cu prioritate mai mică pentru aşi relua propria execuție. Alt factor care poate afecta dispecerul de fire de execuție, este modul în care sistemul de operare implementează multitasking. De aceea, doar prin asignarea de priorități mai mici, mai mari, nu va influența neapărat rapiditatea cu care un fir rulează. Prioritatea înseamnă un acces probabil mai mic sau mai mare la CPU.

Atunci când un fir de execuție porneşte, prioritatea sa este egală cu cea a thread-ului părinte. Valoarea nivelului de prioritate trebuie să fie între MIN_PRIORITY şi MAX_PRIORITY. În mod normal, aceste valori sunt 1 şi 10. Pentru a returna un fir cu prioritate implicită, avem NORM_PRIORITY ce înseamnă 5. Aceste valori sunt constante în clasa *Thread*. Se poate obține prioritatea actuală a unui fior de execuție apelând metoda *getPriority()*:

```
final int getPriority()
```

Exemplul următor demonstrează folosirea a două fire de prioritate diferită. Metoda *run()* conține o bulcă while care contorizează iterații. Această bulcă se opreşte când numărul de iterații depăşeşte 1000000 sau variabila *stop* a fost setată pe *true*. Inițial *stop* este setată pe *false*, dar primul thread care termină de iterat, o va seta pe *true*. Evident aceasta va face ca şi al doilea thread să încheie bucla. La fiecare parcurgere a buclei, variabila string, *currentName*, este comparată cu firul aflat în execuție. Dacă este diferită, se realizează modificarea acestei variabile. Aceasta permite vizualizarea accesării CPU a unui anumit thread.

```
class Priority implements Runnable
{
    int count;
    Thread thread;
    static boolean stop = false;
    static String currentName;
    //Constructorul unui nou fir
    //acesta nu porneşte thread-ul
    Priority(String name)
    {
        thread = new Thread(this, name);
        count = 0;
        currentName = name;
    }
    // incepe execuția unui nou fir
    public void run()
    {
        System.out.println(thread.getName() + " starting.");
```

```
        do
        {
            count++;
            if(currentName.compareTo(thread.getName()) != 0)
            {
                currentName = thread.getName();
                System.out.println("In " + currentName);
            }
        } while(stop == false && count < 10000000);
        stop = true;
        System.out.println("\n" + thread.getName() +
        " terminating.");
    }
}
class PriorityDemo
{
    public static void main(String args[])
    {
        Priority mt1 = new Priority("High Priority");
        Priority mt2 = new Priority("Low Priority");
        // mt1 primeşte o prioritate mai mare decât mt2
        mt1.thread.setPriority(Thread.NORM_PRIORITY+2);
        mt2.thread.setPriority(Thread.NORM_PRIORITY-2);
        //pornesc ambele tread-uri
        mt1.thread.start();
        mt2.thread.start();
        //aşteptarea terminării ambelor thread-uri
        try
        {
            mt1.thread.join();
            mt2.thread.join();
        }
        catch(InterruptedException exc)
        {
            System.out.println("Main thread interrupted.");
        }

        System.out.println("\nHigh priority thread counted to " +
        mt1.count);
        System.out.println("Low priority thread counted to " +
        mt2.count);
    }
}
```

Efectul rulării acestui program este:

```
High Priority starting.
In High Priority
Low Priority starting.
In Low Priority In High Priority
In Low Priority
In Low Priority
In High Priority
…
In High Priority
In High Priority
In Low Priority
In High Priority
In High Priority
In Low Priority
In High Priority
High Priority terminating.

Low Priority terminating.

High priority thread counted to 10000000
Low priority thread counted to 1194749
```

În acest mod putem observa că pentru acest exemplu, thread-ul cu prioritate mai mare a avut majoritatea timpului CPU. Acest rezultat va depinde de sistemul de operare şi maşina pe care va rula.

Sincronizarea

Atunci când folosim mai multe fire de execuție, este necesar uneori, să coordonăm fluxul logic, acest proces numindu-se *sincronizare*. Cel mai simplu motiv pentru această sincronizare, este ca două sau mai multe fire să aibă acces la o resursă comună, dar doar un singur fir să acceseze la un moment dat acea resursă. De exemplu scrierea într-un fişier, efectuată din două fire de execuție trebuie controlată. Aceasta se realizează astfel: un fir este pus în stare de aşteptare până când firul care are acces la resursă termină acțiunea, urmând ca firul suspendat să îşi reia execuția.

Sincronizarea în Java este realizată prin intermediul conceptului de *monitor*, ce controlează accesul la un obiect. Monitorul funcționează implementând conceptul de blocare. Când un obiect este blocat de un fir de execuție, nici un alt fir de execuție nu are acces la acel obiect. Când firul de execuție eliberează acel lock, obiectul devine disponibil pentru celelalte fire de execuție.

Toate obiectele în Java au un monitor. Această caracteristică este implementată în limbaj. De aceea toate obiectele pot fi sincronizate. Acest lucru se va realiza prin cuvântul cheie *synchronized* şi alte câteva metode pe care toate obiectele le au. Există două metode prin care se poate sincroniza fluxul logic al unui program.

Folosirea metodelor sincronizate

Se poate sincroniza accesul la o metodă folosind cuvântul cheie *synchronized*. Atunci când acea metodă este apelată , firul de execuție în care s-a făcut apelul intră în obiectul monitor, care blochează obiectul. În timpul blocării, nici un alt thread nu poate intra în metodă, sau accesa orice metodă sincronizată. Atunci când firul de execuție revine din metodă, se realizează deblocarea.

Următorul program demonstrează folosirea acestei metode de sincronizare:

```
//clasa ce calculează o sumă de elemente
//si conține metoda sincronizata
class SumArray
{
    private int sum;
    //metoda are specificatorul de sincronizare
    synchronized int sumArray(int nums[])
    {
        sum = 0; // resetarea sumei
        for(int i=0; i<nums.length; i++)
        {
            sum += nums[i];
            System.out.println("Running total for " +
            Thread.currentThread().getName() +
            " is " + sum);
            try
            {
                Thread.sleep(15); //timp in care
                //se poate schimba de la un fir la altul
            }
            catch(InterruptedException exc)
            {
                System.out.println("Main thread interrupted.");
            }
        }
        return sum;
    }
}
//clasa care se ocupa de thread
class MyThread implements Runnable
{
    Thread thread;
    static SumArray sa = new SumArray();
    int a[];
    int answer;
    // Constructorul unui nou fir
    MyThread(String name, int nums[])
    {
        thread = new Thread(this, name);
        a = nums;
```

```
        thread.start(); //pornesc thread-ul
    }
    public void run()//incepe execuția noului fir
    {
        int sum;
        System.out.println(thread.getName() + " starting.");
        answer = sa.sumArray(a);
        System.out.println("Sum for " + thread.getName() +
        " is " + answer);
        System.out.println(thread.getName() + " terminating.");
    }
}
class SyncDemo
{
    public static void main(String args[])
    {
        int a[] = {1, 2, 3, 4, 5};
        MyThread mt1 = new MyThread("Child #1", a);
        MyThread mt2 = new MyThread("Child #2", a);
    }
}
```

Acest exemplu este compus din trei clase. Prima şi anume `SumArray` conține metoda `sumArray` ce realizează însumarea elementelor dintr-un şir. Acest şir este pasat ca parametru al metodei. A doua clasă este, `MyThread` cea care se ocupă de firul de execuție propriu zis. Firul de execuție va apela metoda `sumArray` prin intermediul obiectului `sa` de tip `SumArray`. Acesta este declarat *static* în clasa `MyThread`. Pe metoda de *run()* a firului de execuție se va rula această metoda de însumare a elementelor unui şir. În clasa `SyncDemo`, în metoda *main()* este creat un şir şi anume *a*, şi două fire de execuție ce au ca parametru acelaşi şir. Cu alte cuvinte, aceeaşi metodă `sa.sumArray(a);` va fi apelată în ambele fire, creând o concurență de acces la suma rezultat. Suma nu va fi însă afectată, deoarece firele vor executa succesiv metoda `sumArray`.

Rezultatul este:

```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #1 is 3
Running total for Child #1 is 6
Running total for Child #1 is 10
Running total for Child #1 is 15
Running total for Child #2 is 1
Sum for Child #1 is 15
Child #1 terminating.
Running total for Child #2 is 3
Running total for Child #2 is 6
Running total for Child #2 is 10
Running total for Child #2 is 15
Sum for Child #2 is 15
Child #2 terminating.
```

Aceasta înseamnă că execuția a avut loc conform aşteptărilor şi sumele calculate sunt cele prevăzute. Dar dacă eliminăm cuvântul cheie **`synchronized`** din cadrul metodei, iată ce obținem:

```
Child #2 starting.
Running total for Child #2 is 1
Child #1 starting.
Running total for Child #1 is 1
Running total for Child #2 is 3
Running total for Child #1 is 5
Running total for Child #2 is 8
Running total for Child #1 is 11
Running total for Child #2 is 15
Running total for Child #1 is 19
Running total for Child #1 is 24
Running total for Child #2 is 29
Sum for Child #1 is 29
Child #1 terminating.
Sum for Child #2 is 29
Child #2 terminating.
```

Evident, rezultatele pot varia în funcție de maşină, sistem de operare, etc.

Blocul synchronized

Deşi crearea metodelor sincronizate, în cadrul claselor, este un mod eficient şi uşor de sincronizare, nu funcționează pentru orice caz. De exemplu, să presupunem că dorim accesul sincronizat la o metodă care nu este declarată ca atare. Acest lucru se poate întâmpla în cazul în care clasa a fost creată de altcineva şi nu avem acces la codul sursă. Pentru a accesa obiecte care nu au fost anterior sincronizate, avem la dispoziție blocuri sincronizate:

```
synchronized(object)
{
      //instrucțiunile ce trebuie sincronizate
}
```

Aici `object` este referința către obiectul de sincronizat. Un bloc sincronizat asigură faptul că apelul unei metode membră a obiectului `object`, va avea loc într-un monitor al firului de execuție apelant. De exemplu putem reedita programul mai sus considerând că metoda a fost declarată fără cuvânt cheie **`synchronized`**:

```
int sumArray(int nums[])
```

Ceea ce se modifică, este în cadrul metodei run() a clasei Thread:

```
public void run()
    {
        synchronized(sa)
        {
            int sum;
            System.out.println(thread.getName() + " starting.");
            answer = sa.sumArray(a);
            System.out.println("Sum for " + thread.getName() +
            " is " + answer);
            System.out.println(thread.getName() + " terminating.");
        }
    }
```

În acest fel se asigură un calcul predictiv al sumei format din elementele lui *sa*.

Comunicarea între fire de execuție

Să considerăm următoarea situație. Un fir numit T execută o metodă sincronizată şi are nevoie de o resursă R, care este temporar indisponibilă. Ce ar trebui să facă T? Dacă T intră într-o buclă de aşteptare a lui R, blochează obiectul prevenind alte fire să o acceseze. Această abordare este consumatoare de timp şi poate duce la deadlock-uri adică blocare permanentă. O altă soluție ar fi ca T să elibereze temporar controlul obiectului, permițând altui fir să ruleze. Când R devine disponibilă, T poate fi notificat şi îşi poate relua execuția. Acest tip de abordare, presupune existența unei comunicări între fire diferite. În Java aceasta se face prin metodele *wait()*, *notify()* şi *notifyAll()*.

Aceste metode aparțin tuturor obiectelor, deoarece sunt implementate în clasa Object. Aceste metode pot fi apelate dintr-o metodă sincronizată. Atunci când un thread este blocat temporar, apelează *wait()*. Aceasta va face ca firul să intre în *sleep*, şi monitorul pentru acel obiect să fie eliberat, permițând altui fir să folosească obiectul. Mai târziu, firul ce era oprit, este trezit, atunci când un alt fir care va intra în acelaşi monitor, apelează metoda *notify()*, sau *notifyAll()*. Un apel la metoda *notify()* va reporni firul oprit.

Iată diversele forme ale metodei *wait()*:

```
final void wait( ) throws InterruptedException
final void wait(long millis) throws InterruptedException
final void wait(long millis, int nanos) throws InterruptedException
```

Ultimele două forme ale metodei semnifică aşteptarea până când apare o notificare sau până când o perioada de timp, dată de parametrii, trece. Mai jos avem formele funcțiilor de notificare:

```
final void notify( )
final void notifyAll( )
```

Iată un exemplu de folosire a *wait()* şi *notify()*:

```
class TicTac
{
    synchronized void tic(boolean running)
    {
        if(!running)
        { //opresc metoda
            notify(); //anunţ celălalt thread
            return;
        }
        System.out.print("Tic ");
        notify(); //il las pe tac sa ruleze
        try
        {
            wait(); // aştept pe tac
        }
        catch(InterruptedException exc)
        {
            System.out.println("Thread interrupted.");
        }
    }
    synchronized void tac(boolean running)
    {
        if(!running)
        { // opresc pe tac
            notify(); // anunţ pe tic
            return;   // adică celălalt fir
        }
        System.out.println("Tac");
        notify(); // il las pe tic sa ruleze
        try
        {
            wait(); // aştept ca tic sa ruleze
        }
        catch(InterruptedException exc)
        {
            System.out.println("Thread interrupted.");
        }
    }
}
class MyThread implements Runnable
{
    Thread thread;
    TicTac ttOb;
    // un nou fir de execuţie
    MyThread(String name, TicTac tt)
    {
```

```
            thread = new Thread(this, name);
            ttOb = tt;
            thread.start(); // pornesc firul
        }
        // începe execuţia firului
        public void run()
        {
            if(thread.getName().compareTo("Tic") == 0)
            {
                // sunt in thread-ul tic
                for(int i=0; i<5; i++) ttOb.tic(true);
                //la sfarsit opresc tic
                ttOb.tic(false);
            }
            else
            {
                // sunt in thread-ul tac
                for(int i=0; i<5; i++) ttOb.tac(true);
                 //la sfarsit opresc tac
                ttOb.tac(false);
            }
        }
    }
    class ThreadsDemo
    {
        public static void main(String args[])
        {
            TicTac tt = new TicTac();
            MyThread mt1 = new MyThread("Tic", tt);
            MyThread mt2 = new MyThread("Tac", tt);

            try
            {
                mt1.thread.join();
                mt2.thread.join();
            }
            catch(InterruptedException exc)
            {
                System.out.println("Main thread interrupted.");
            }
        }
    }
```

În acest exemplu există trei clase, `TicTac`, `MyThread`, `ThreadsDemo`. Prima clasa, conține două metode sincronizate:

```
synchronized void tic(boolean running)
synchronized void tac(boolean running)
```

Acestea, au următoarea logică:

```
if(!running)
        { //opresc metoda
            notify(); //anunț celălalt thread
            return;
        }
        System.out.print("Tic ");
        notify(); //il las pe tac sa ruleze
        wait(); // aştept pe tac
```

Dacă parametrul transmis nu este încă *false* (mai am drept să rulez metoda) atunci afișez *Tic* şi dau dreptul lui *Tac*, urmând să aştept până când thread-ul ce rulează *Tac,* să mă înştiințeze că a rulat în monitor. Dacă parametrul `running` este *false*, atunci anunț celălalt fir de execuție şi părăsesc metoda.

Acelaşi lucru are loc şi pentru metoda *tac*, evident referitor la thread-ul *Tic*.

Clasa `MyThread`, ce implementează interfața *Runnable*, reprezintă cele două fire de execuție suport pentru metodele *tic()* şi *tac()*. În firul de execuție pentru *Tic* se apelează de cinci ori metoda *tic()* cu parametru *true*, şi ultima dată cu parametru *false*, pentru a permite deblocarea firului care executa *tac()*. Acelaşi lucru este valabil viceversa pentru *Tac*.

În clasa `ThreadsDemo` se creează cele două fire de execuție, se lansează şi se aşteaptă terminarea lor. Iată ce se întâmplă la rularea programului:

```
Tic Tac
Tic Tac
Tic Tac
Tic Tac
Tic Tac
```

Dacă modificăm însă, metodele `tac()` şi `tic()` eliminând metodele de comunicare şi anume `wait()` şi `notify()` rezultatul poate fi:

```
Tic Tic Tic Tic Tic Tac
Tac
Tac
Tac
Tac sau
Tac
Tac
Tac
Tac
Tac
Tic Tic Tic Tic Tic
```

Suspendarea, reluarea şi oprirea firelor de execuţie

Uneori este util să suspendăm execuția unui thread. De exemplu, un fir de execuție poate fi folosit pentru afişarea orei. Dacă utilizatorul nu doreşte vizualizarea orei, acest fir va fi suspendat. El va putea fi repornit (*resume*) ulterior. Metodele ce realizează acest lucru sunt:

```
final void resume( )
final void suspend( )
final void stop( )
```

Totuşi folosirea acestora este îngrădită, de unele probleme cauzate de *suspend()* în trecut. De exemplu, dacă un thread obține un lock pe o zonă critică, şi el este suspendat, acele lock-uri sunt eterne. În acest timp, alte fire de execuție aşteaptă deblocarea lor. Metoda *resume()* este de asemenea învechită. Nu cauzează probleme, dar nu poate fi folosită fără metoda *suspend()*. De asemenea metoda *stop()* a fost considerată învechită.

Următorul exemplu reprezintă un mod în care se pot folosi aceste funcții:

```
class MyThread implements Runnable
{
    Thread thread;
    //volatile adică variabila poate fi
    //modificata de alte parti ale programului
    //cum ar fi un thread
    volatile boolean suspended;
    volatile boolean stopped;
    MyThread(String name)
    {
        thread = new Thread(this, name);
        suspended = false;
        stopped = false;
        thread.start();
    }
    // entry point pentru un fir de execuție
    public void run()
    {
        System.out.println(thread.getName() + " starting.");
        try
        {
            for(int i = 1; i < 1000; i++)
            {
                System.out.print(i + " ");
                //din 10 in 10 are loc o pauza
                if((i%10)==0)
                {
                    System.out.println();
                    Thread.sleep(250);
                }
```

```
                //bloc sincronizat pentru verificarea
                 //stopped sau suspended
                 synchronized(this)
                 {
                     while(suspended)
                     {
                         wait();
                     }
                     //in cadrul opririi se iese din fir
                     if(stopped) break;
                 }
             }
         }
         catch (InterruptedException exc)
         {
             System.out.println(thread.getName() + " interrupted.");
         }
         System.out.println(thread.getName() + " exiting.");
     }
     // stop firul de execuţie
     synchronized void mystop()
     {
         stopped = true;
         // firul suspendat este oprit
         suspended = false;
         //anunţ firul care este suspendat
         notify();
     }
     //suspendarea firului
     synchronized void mysuspend()
     {
         suspended = true;
     }
     //Reluarea firului
     synchronized void myresume()
     {
         suspended = false;
         //anunţ firul care este suspendat
         notify();
     }
}
class SuspendDemo
{
     public static void main(String args[])
     {
         MyThread ob1 = new MyThread("My Thread");
         try
         {
```

```
            Thread.sleep(1000); //pauză pe firul principal
            ob1.mysuspend(); //suspend firul ob1
            System.out.println("Suspending thread.");
            Thread.sleep(1000);
            ob1.myresume(); //reiau firul ob1
            System.out.println("Resuming thread.");
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Suspending thread.");
            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Resuming thread.");
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Stopping thread.");
            ob1.mystop();
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread Interrupted");
        }
        try
        {
            ob1.thread.join();
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```

Clasa `MyThread`, defineşte două variabile booleane, `suspended` şi `stopped`. Metoda *run()* conține blocul sincronizat ce verifică variabila `suspended`. Dacă este *true* atunci metoda *wait()* este apelată şi suspendă execuția firului. Metoda care stabileşte valoarea variabilei la *true,* este *mysuspend()*. Pentru a relua firul, avem *myresume()*, care setează variabila `suspended` pe *true* şi apelează *notify()*, pentru a relua execuția firului. Ultima metodă este *mystop()* şi conține paşii care duc la oprirea firului de execuție. În clasa `SuspendDemo`, metodele descrise sunt apelate în alternanță pentru a opri/relua execuția firului de câteva ori. Rezultatul rulării acestui program este:

```
My Thread starting.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Suspending thread.
```

```
Resuming thread.
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Suspending thread.
Resuming thread.
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120
Stopping thread.
My Thread exiting.
Main thread exiting.
```

Grupuri de fire de execuție

Fiecare fir de execuție este membru al unui grup de fire. Un grup de thread-uri oferă un mecanism pentru colectarea multiplelor fire într-un singur obiect pentru a manipula aceste fire. De exemplu, se pot suspenda toate firele printr-o singură instrucțiune.

Dacă vom crea un fir fără a specifica grupul din care face parte, automat va fi pus în acelaşi grup cu cel al firului părinte. Acest grup se mai numeşte şi *current thread group*. Atunci când o aplicație este pornită, Java creează un *TreadGroup* numit *main*. Pentru a crea un grup separat de acesta, explicit, avem la îndemână următoarele construcții:

```
public Thread(ThreadGroup group, Runnable target)
public Thread(ThreadGroup group, String name)
public Thread(ThreadGroup group, Runnable target, String name)
```

Mai jos există un mod de folosire a acestor constructori:

```
ThreadGroup myThreadGroup = new ThreadGroup("My Group of Threads");
Thread myThread = new Thread(myThreadGroup, "a thread for my group");
```

Pentru a prelua grupul unui fir de execuție avem metoda *getThreadGroup()*:

```
theGroup = myThread.getThreadGroup();
```

Iată un mod de folosire a acestei informații:

```
public static void listCurrentThreads()
    {
        ThreadGroup currentGroup =
        Thread.currentThread().getThreadGroup();
        int numThreads = currentGroup.activeCount();
        Thread[] listOfThreads = new Thread[numThreads];
```

```
        currentGroup.enumerate(listOfThreads);
        for (int i = 0; i < numThreads; i++)
        {
            System.out.println("Thread #" + i + " = "
                   + listOfThreads[i].getName());
        }
    }
```

Această funcție preia numărul de fire din grup si le afişează.

Alte metode

Clasa ThreadGroup conține câteva atribute şi metode dintre care:

- *getMaxPriority* şi *setMaxPriority* pentru manipularea priorității
- *getDaemon* şi *setDaemon* pentru controlul acestori tipuri de fire
- *getName* pentru preluarea numelui
- *getParent* şi *parentOf* pentru manipularea numelui firului părinte

Iată, spre exemplu, cum se poate stabili prioritatea maximă, atât pe fir cât şi pe grup:

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

        ThreadGroup groupNORM = new ThreadGroup(
                                "A group with normal priority");
        Thread priorityMAX = new Thread(groupNORM,
                             "A thread with maximum priority");

        // set Thread's priority to max (10)
        priorityMAX.setPriority(Thread.MAX_PRIORITY);

        // set ThreadGroup's max priority to normal (5)
        groupNORM.setMaxPriority(Thread.NORM_PRIORITY);

        System.out.println("Group's maximum priority = "
                           + groupNORM.getMaxPriority());
        System.out.println("Thread's priority = "
                           + priorityMAX.getPriority());
    }
}
```