Threads – Sintaxa / Stari / Sincronizare (continuat)

1. Definire:

Varianta 1 – prin derivare din clasa **Thread**	Varianta 2 – prin implementarea interfetei **Runnable**

class Fir extends Thread{
public void run(){…}
}
	class Fir extends Ceva implements Runnable{
	public void run(){…}
	}

2. Instantiere:

Varianta 1 – prin derivare din clasa **Thread**	Varianta 2 – prin implementarea interfetei **Runnable**
Fir f = new Fir();	Fir obf = new Fir(); Thread f = new Thread (obf);

3. Gata de rulare:

Varianta 1 – prin derivare din clasa **Thread**	Varianta 2 – prin implementarea interfetei **Runnable**
f.start();	f.start();

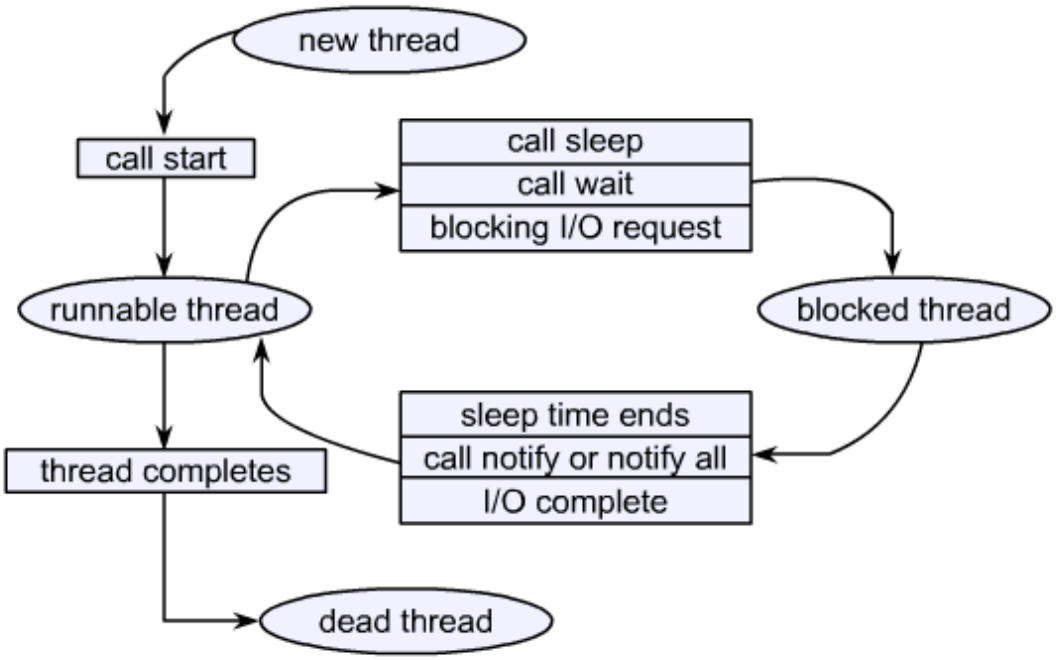
4. Metode specifice clasei Thread:

Varianta 1 – prin derivare din clasa **Thread**	Varianta 2 – prin implementarea interfetei **Runnable**

public void run(){
this.sleep();
}
	public void run(){
	Thread t = Thread.currentThread();
	…
	t.sleep();
	}

5. Starile unui fir de executie:

- **Creat** (**new thread**) – utilizand operatorul **new**
- **Gata de executie** (**runnable thread**) – dupa apelul metodei **start()**
- **Suspendat** (**blocked thread**) – daca un fir de executie apeleaza metoda **sleep()** sau **wait()**
- **Terminat** (**dead thread**) **–** dupa ce se termina metoda **run()**



Exemplu: Fir1.java

6. Sincronizarea firelor de executie:

Problema sincronizarii apare in cazul de:

- **A. Concurenta** (se incearca utilizarea acelorasi resurse)
- **B. Cooperare** (se incearca interschimb de informatii)

A. Rolul sincronizarii in acest caz este de a asigura accesul exclusiv la resursele comune. O secventa de cod pentru care trebuie sa se execute accesul exclusiv (un singur proces/fir de executie executa la un moment dat secventa respectiva de cod) se numeste **regiune critica**.

A1. Mai multe fire de executie pot modifica acceasi resursa comuna
(FirNe2.java => FirSi2.java)

A2. Mai multe fire de excutie pot executa in paralel metodele aceluiasi obiect
(FirNe3.java => FirSi3.java)

* Aparent similar dar cu rezolvari diferite.

B. Sincronizarea inseamna in acest caz ca procesele (firele de executie), care se pot executa cu viteze diferite, trebuie sa se astepte unele pe altele pana cand pot sa-si transmita informatiile de care au nevoie.

ProducatorConsumator.java

Probleme speciale:

- Metodele **wait()**, **notify()** si **notifyAll()** sunt a clasei Object si nu a clasei Thread. Metoda wait() poate fi apelata **DOAR** dintr-o metoda sincronizata.
- De multe ori in programare este necesar sa se asocieze unui zavor mai multe conditii/evenimente care ar putea sa produca blocarea unor fire de executie care utilizeaza obiectul respectiv (resursa comuna). *Se pune problema cum se face deosebirea intre aceste evenimente. Solutia consta din asocierea unor conditii logice la evenimentele respective.* O metoda **wait()** se apeleaza intr-un ciclu de forma:

```
while (! conditie) {
   this.wait();
}
```

- Pentru variable ce formeaza resursa comuna se recomanda utilizarea cuvantului cheie **volatile**. Atributul este folosit pentru a indica faptul ca o variabila poate sa-si modifice valoarea datorita unei metode nesincronizate. Orice modificare trebuie scrisa cat mai

repede in memorie si nu pastrata intr-un registru in microprocesor pentru optimizare.

- Daca un alt fir de executie (inclusiv cel pe care se executa metoda void main(String[] args)) trebuie sa astepte un alt fir de executie poate utiliza metoda **join()** :

```
class Fir extends Thread {
 private double rezultat;

 public void run() {  this.rezultat = this.calculeaza(); }

 public double calculeaza() {
  double valoare;
  //... o galagie de calcule
  return valoare;
 }

 public double getSolutie() { return this.rezultat; }
}

class ProgMainJoin {
  public static void main(String[] args) {
    Fir f = new Fir();
    f.start();
    try {
      f.join();
      System.out.println("Rezultat: "+f.getSolutie());
    } catch(InterruptedException ie) {ie.printStackTrace();}
  }
}
```

- Utilizarea **prioritatilor** face ca firul de executie cu prioritate mai mare sa castige dreptul de utilizare a microprocesorului mai des:

```
class FirExemplu extends Thread {
  String mesaj;
  public FirExemplu(String m) {this.mesaj = m;}
  public void run() {
     while (true) {
       System.out.println(this.mesaj);
     }
  }
}

class ProgMainFir1 {
 public static void main(String[] args) {
   FirExemplu f1 = new FirExemplu("f1");
   FirExemplu f2 = new FirExemplu("f2");

    f1.start();
```

```
    f2.start();
 }
}

class ProgMainFir2 {
 public static void main(String[] args) {
   FirExemplu f1 = new FirExemplu("f1");
   FirExemplu f2 = new FirExemplu("f2");

   f1.start();

    Thread curent = Thread.currentThread();
    f2.setPriority(curent.getPriority() + 1);
    f2.start();
 }
}
```

- Metoda **yield()** permite sistemului de operare sa comute controlul de la firul de executie curent la un alt fir de executie cu aceeasi prioritate:

```
//modificarea metodei run() pentru exemplu anterior
public void run() {
 while (true) {
   System.out.println(mesaj);
   this.yield();//daca firele de executie au aceeasi prioritate, vor alterna
 }
}
```

- Pentru setarea prioritatilor (setPriority() si getPriority()) trebuie rulat exemplul **Priority.java**
- Pentru exemplu de utilizare metodele join(), join(...), isAlive() si interrupt() trebuie rulat **SimpleThreads.java**