Curs 8 PPOO

Conf. univ. dr. Cristian CIUREA

Departamentul de Informatică și Cibernetică Economică

cristian.ciurea@ie.ase.ro

Java fundamentals

- Fire de execuție
- Procese

Fire de execuție

- Simultaneitatea execuției programelor în Java este abilitatea de a rula mai multe programe sau mai multe părți ale unui program în paralel.
- În cazul în care o activitate consumatoare de timp poate fi realizată în mod asincron sau în paralel, aceasta va îmbunătăți obținerea rezultatelor și interactivitatea programului.
- Un calculator modern dispune de mai multe nuclee la nivelul procesorului, ceea ce înseamnă că abilitatea de a mobiliza aceste multi-core-uri poate fi cheia pentru execuția cu succes a unei aplicații cu volum mare de date.

Fire de execuție

- "**Multithreading**" înseamnă capacitatea unui program de a executa mai multe secvențe de cod în același timp.
- O astfel de secvență de cod se numește fir de execuție sau **thread**.
- Limbajul Java suportă multithreading prin clase disponibile în pachetul **java.lang**.

Fire de execuție

- Un **fir de execuție (thread)** este un flux de control în cadrul unui program.
- Un **proces** poate avea mai multe fire de execuție.

Solution 1 – extending Thread	Solution 2 – implementing Runnable

class **NewThread** extends **Thread**{
public void run(){…}
}
	class **NewThread** implements **Runnable**{
	public void run(){…}
	}

Fire de execuție

Instantiating threads

Solution 1 – extending Thread	Solution 2 – implementing Runnable
NewThread f = new **NewThread** ();	**NewThread** obf = new **NewThread** (); Thread f = new Thread (obf);

Executing threads

Solution 1 – extending Thread	Solution 2 – implementing Runnable
f.start();	f.start();

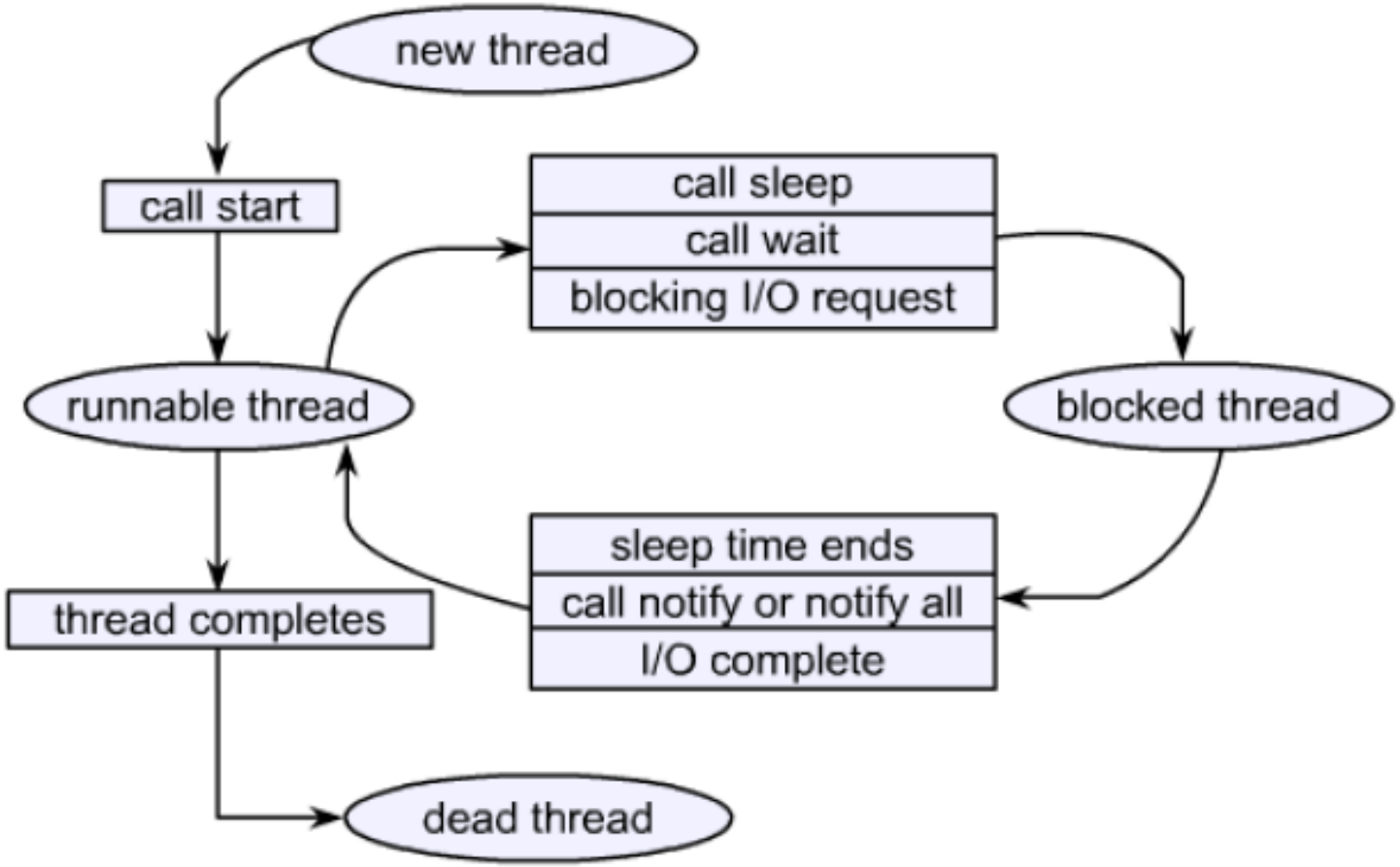
Fire de execuție

Controlling threads

Solution 1 – extending Thread	Solution 2 – implementing Runnable

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

Fire de execuție



Fire de execuție

- **new thread**: atunci când se instanțiază cu *new* un nou fir de execuție;
- **runnable thread**: după apelul metodei **start()**;
- **blocked thread**: dacă se apelează **sleep()** sau **wait()**;
- **dead thread**: după ce metoda **run()** s-a terminat.

Fire de execuție

- Există două tipuri de fire de executie într-o aplicație - **user threads** și **daemon threads**. Când pornim o aplicație, main() este primul fir de executie de tip utilizator creat și putem crea mai multe fire de tip utilizator, precum și fire de tip daemon. Când toate firele de executie de tip utilizator sunt executate, JVM încheie programul.

Fire de execuție

- Putem seta priorități diferite pentru diferite fire de executie, dar nu se garantează faptul că thread-ul cu prioritate mai mare se va executa mai întâi față de altul cu prioritate inferioară.
- Planificatorul de fire de execuție este o parte a sistemului de operare și atunci când este pornit un fir, execuția este controlată de **Thread Scheduler**, iar JVM nu are niciun control asupra execuției acestuia.

Fire de execuție

- Un fir de execuție de tip **daemon** este un fir care nu împiedică ieșirea din program când acesta se termină, ci firul de execuție continuă să ruleze. Un exemplu de fir de execuție de tip daemon este **garbage collector**.
- Se poate apela metoda **setDaemon(boolean)** pentru a modifica proprietățile unui fir de execuție înainte ca firul să pornească.

Fire de execuție

- Metoda **main()** are propriul fir de execuţie. Prin apelul **start()** se cere JVM crearea și pornirea unui nou fir de execuție. Din funcția **start()** se va ieși imediat. Firul de execuție corespunzător metodei **main()** își va continua execuția independent de noul fir de execuție creat.
- Implementările Java depind de platformă/sistem de operare. Un program Java care folosește fire de execuție poate avea comportamente diferite la execuții diferite pentru aceleași date de intrare.

Fire de execuție

Sincronizarea firelor de execuție:

- **concurență**: utilizarea acelorași resurse;
- **cooperare**: interschimb de date.

Controlul firelor de execuție:

- metodele **wait()**, **notify()** și **notifyAll()** moștenite din clasa *Object*;
- metoda **join()** în clasa *Thread*;
- metoda **setPriority()** în clasa *Thread*;
- metodele statice **yield()** și **sleep()** în clasa *Thread*.

Fire de execuție

Controlul firelor de execuție:

- metoda **setPriority()** permite setarea unei priorități de la 1 la 5 sau de la 1 la 10 pentru firul curent de execuție;
- metoda **join()** permite ca firul curent de execuție să aștepte alt fir de execuție să se termine (firul curent de execuție se unește cu celălalt);
- metoda **yield()** întrerupe firul curent de execuție și oferă controlul unui alt fir de execuție având aceeași prioritate;
- metoda **sleep()** întrerupe firul curent de execuție pentru o perioadă de timp.

Fire de execuție

- Java implementează mecanismul de excludere mutuală (**mutual exclusion**) prin specificarea metodelor care partajează variabile ca fiind **synchronized**.
- Orice fir de execuție care încearcă să acceseze o metodă **synchronized** a unui obiect, atât timp cât metoda este utilizată de un alt fir de execuție, este blocat până când primul fir de execuție părăsește metoda.

Fire de execuție

- Tot o problemă de excludere mutuală este exemplul producător-consumator. Producătorul furnizează date pe care consumatorul le utilizează mai departe. Dacă vitezele de producere și consum diferă se utilizează metodele **wait(), notify()** și **notifyAll()** ale clasei **Object**.
- Apelul lui **wait()** va trece obiectul apelat în starea Blocked. El rămâne blocat până la apelul unei metode **notify()** sau **notifyAll()** pentru același obiect. Condiția necesară pentru a se apela una dintre aceste metode este ca apelul lor să se facă în interiorul metodelor **synchronized**.
- Pentru metoda **wait()** se poate specifica o durată maximă de așteptare. În acest caz, firul de execuție rămâne blocat până când timpul expiră sau un alt fir de execuție apelează **notify()**.

Fire de execuție

Concurența / simultaneitatea:

- permite ca metodele **sincronizate** să acceseze resurse comune;
- fiecare obiect are o singură blocare (gestionată de JVM);
- variabilele comune sunt declarate ca **volatile** (modificările sunt șterse din memorie);
- utilizează metodele **wait()**, **notify()** și **notifyAll()** moștenite din clasa Object.

Fire de execuție

Concurența / simultaneitatea:

- Urmărește firele de execuție blocate;
- metodele **wait()**, **notify()** și **notifyAll()** trebuie să fie apelate dintr-un context sincronizat;
- implementarea specifică a metodei **wait()**:

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

Procese

- Un proces rulează independent și izolat de alte procese. El nu poate avea acces direct la datele partajate din alte procese. Resursele procesului, de exemplu, memorie și timp CPU, sunt alocate acestuia prin intermediul sistemului de operare.
- Un fir de execuție este un proces denumit “ușor” (**lightweight**). El are propria sa stivă de apel, dar poate avea acces la date partajate ale altor fire de execuție în cadrul aceluiași proces. Fiecare fir are propria sa memorie cache.

Procese

- Un flux de ieșire **PipedOutputStream** poate fi conectat la un flux de intrare **PipedInputStream** pentru a crea o conductă/țeavă de comunicație. În mod obișnuit, datele sunt scrise într-un obiect **PipedOutputStream** prin intermediul unui fir de execuție și sunt citite dintr-un **PipedInputStream** conectat printr-un alt fir de execuție. Încercarea de a utiliza ambele obiecte dintr-un singur fir de execuție nu este recomandată, deoarece ar putea bloca firul. Se consideră că țeava este ruptă dacă un fir de execuție care citea octeți de date din fluxul de intrare conectat nu mai este în viață.

Procese

- În cazul în care un fir de execuție citește date partajate, el stochează aceste date în propria sa memorie cache. Un fir de execuție poate reciti datele partajate.
- O aplicație Java rulează în mod implicit într-un singur proces. Într-o aplicație Java se lucrează cu mai multe fire de execuție pentru a realiza **procesarea paralelă** sau **comportament asincron**.

Bibliografie

- [1] Jonathan Knudsen, Patrick Niemeyer – *Learning Java*, *3rd Edition*, O'Reilly.
- [2] http://www.itcsolutions.eu
- [3] http://www.acs.ase.ro
- [4] http://inf.ucv.ro/documents/tudori/laborator8_53.pdf