Fire de execuţie

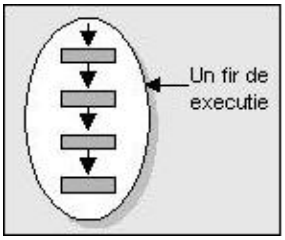
- **Ce este un fir de execuţie ?**
- **Crearea unui fir de execuţie**
- **Ciclul de viaţă**
- **Terminarea firelor de execuţie**
- **Fire de tip "daemon"**
- **Sincronizarea firelor de execuţie**
- **Monitoare**
- **Semafoare**
- **Probleme legate de sincronizare**
- **Gruparea firelor de execuţie**
- **Fluxuri de tip "pipe"**
- **Clasele Timer şi TimerTask**

Ce este un fir de execuţie ?

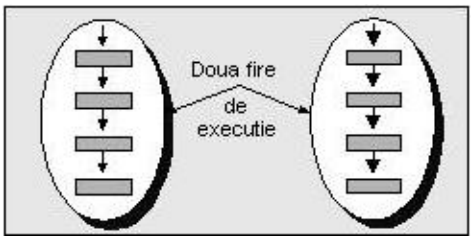
Programare concurentă

Un *fir de execuţie* este o succesiune secvenţială de instrucţiuni care se execută în cadrul unui proces.

Program (proces)



Program (proces)



Proces - **mai multe** fire de execuţie

Procese vs. Fire de execuţie

Asemănări

- Concurenţă
- Planificare la execuţie

Deosebiri

- Un fir de execuţie poate exista doar într-un proces
  Proces uşor, Context de execuţie
- Proces nou: cod + date
  Fir nou : cod

Utilitatea firelor de execuţie

- calcule matematice
- aşteptarea eliberării unei resurse
- desenarea componentelor GUI

Crearea unui fir de execuţie

Orice fir de execuţie este o instanţă a clasei `Thread`

Crearea unui fir de execuţie:

- extinderea clasei `Thread`
- implementarea interfeţei `Runnable`

Interfaţa Runnable

- Defineşte un protocol comun pentru obiecte active
- Conţine metoda **run**
- Este implementată de clasa **Thread**

Extinderea clasei Thread

```
public class FirExcecutie extends Thread {

  public FirExcecutie(String nume) {
    // Apelam constructorul superclasei
    super(nume);
  }

  public void run() {
    //Codul firului de executie
    ...
  }
}
```

Lansarea unui fir

```
// Cream firul de executie
FirExecutie fir = new FirExecutie("simplu");

// Lansam in executie
fir.start();
```

Listing 1: Folosirea clasei Thread

```
class AfisareNumere extends Thread {
  private int a, b, pas;

  public AfisareNumere(int a, int b, int pas) {
    this.a = a;
    this.b = b;
    this.pas = pas;
  }

  public void run() {
    for(int i = a; i <= b; i += pas)
      System.out.print(i + " " );
  }
}

public class TestThread {
  public static void main(String args[]) {
  AfisareNumere fir1, fir2;

  fir1 = new AfisareNumere(0, 100, 5);
  // Numara de la 0 la 100 cu pasul 5

  fir2 = new AfisareNumere(100, 200, 10);
  // Numara de la 100 la 200 cu pasul 10

  fir1.start();
  fir2.start();
  // Pornim firele de executie
  // Ele vor fi distruse automat la terminarea lor
  }
}
```

```
//Secvential...
0 5 10 15 20 25 30 35 40 45 50 55 ...
//Un posibil rezultat
0 100 110 5 10 15 120 130 20 140 25 ...
```

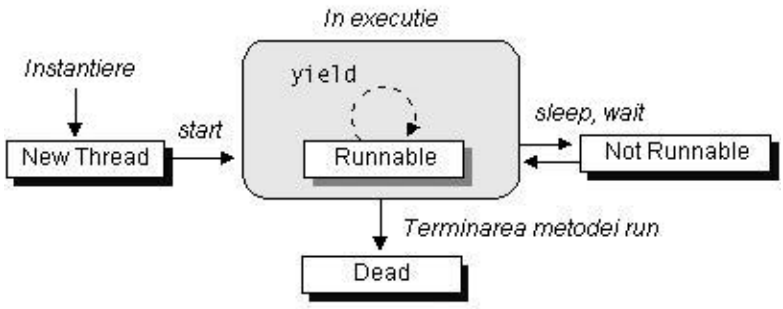
Implementarea interfeţei Runnable

```
class Fir extends Parinte, Thread //Incorect

public class ClasaActiva implements Runnable {
  public void run() {
    //Codul firului de executie
    ...
  }
}
  ...
  ClasaActiva obiectActiv = new ClasaActiva();
  Thread fir = new Thread(obiectActiv);
  fir.start();

  // Sau
public class FirExecutie implements Runnable {
  private Thread fir = null;
  public FirExecutie() {
    fir = new Thread(this);
  }
  public void run() { ... }
}
  ...
  FirExecutie fir = new FirExecutie();
```

Ciclul de viaţă



Starea "New Thread"

```
Thread fir = new Thread(obiectActiv);
// fir se gaseste in starea "New Thread"
```

- Nu are alocate resurse
- Putem apela `start`
- `IllegalThreadStateException`

Starea "Runnable"

```
fir.start();
//fir se gaseste in starea "Runnable"
```

- Alocare resurse
- Planificare la procesor
- Apel `run`

Starea "Not Runnable"

- **sleep**
- `wait` - `notify`
- Operaţii blocante I/O

```
try {
  // Facem pauza de o secunda
  Thread.sleep(1000);
} catch (InterruptedException e) { ...}
```

Starea "Dead"

Firele de execuţie trebuie să se termine **natural**.
Nu *mai* există metoda `stop`.

```
public void run() {
  for(int i = a; i <= b; i += pas)
    System.out.print(i + " " );
}
```

Variabile de terminare

```
public boolean executie = true;
public void run() {
  while (executie) {
    ...
  }
}
```

System.exit termină forţat toate firele de execuţie.

Metoda isAlive

- `true` - "Runnable" sau "Not Runnable"
- `false` - "New Thread" sau "Dead"

```
NumaraSecunde fir = new NumaraSecunde();
// isAlive retuneaza false
// (starea este New Thread)

fir.start();
// isAlive retuneaza true
// (starea este Runnable)

fir.executie = false;
// isAlive retuneaza false
// (starea este Dead)
```

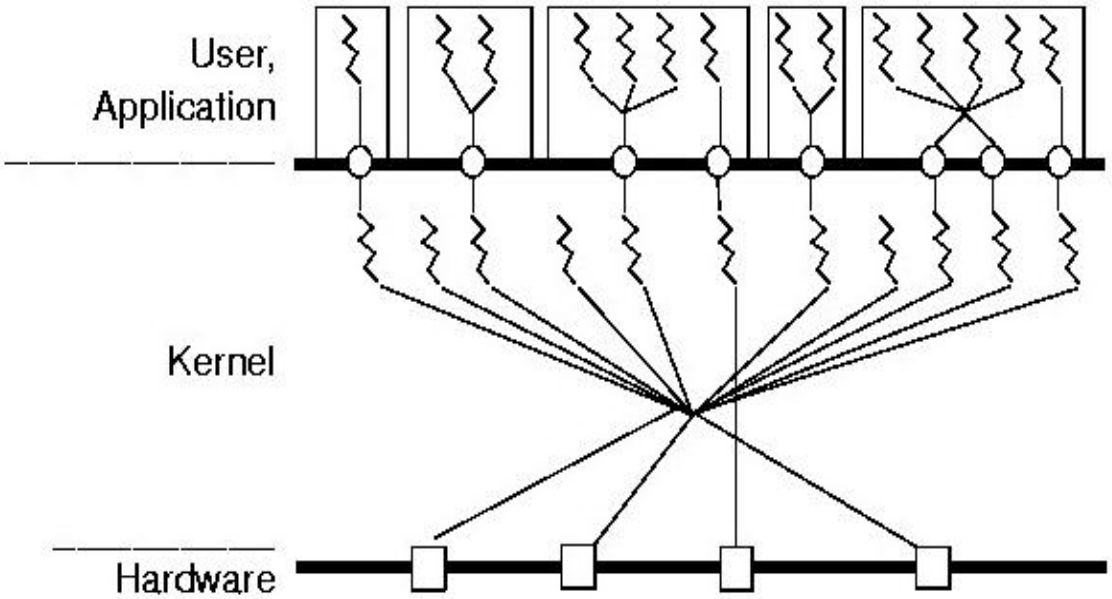
Fire de execuţie de tip "daemon"

Un proces este considerat în execuţie dacă conţine cel puţin un fir de execuţie activ.

Demoni = fire de execuţie care se termină automat la terminarea aplicaţiei.

Metoda **setDaemon**
Permite transformarea unui fir de execuţie în demon sau invers.

Planificarea la execuţie



- Proces
- Fir de execuţie
- Planificator
- Procesor fizic

Priorităţi de execuţie

Modele de lucru

- *Modelul cooperativ*
  partajare **timp**
- *Modelul preemptiv* - "cuante de timp"
  partajare **resurse**

setPriority

```
MAX_PRIORITY = 10;
MIN_PRIORITY = 1;
NORM_PRIORITY= 5;
```

Un fir de execuţie cedează procesorul:

- prioritate mai mare
- metoda sa `run` se termină
- `yield`
- timpul alocat a expirat

Sincronizarea firelor de execuţie

- Partajarea resurselor comune
- Aşteptarea îndeplinirii unor condiţii

Scenariul producător / consumator

```
class Buffer {
  private int number = -1;

  public int get() {
    return number;
  }

  public void put(int number) {
    this.number = number;
  }
}
```

Clasa Producator

```
class Producator extends Thread {
  private Buffer buffer;

  public Producator(Buffer b) {
    buffer = b;
  }

  public void run() {
    for (int i = 0; i < 10; i++) {
      buffer.put(i);
      System.out.println(
        "Producatorul a pus:\t" + i);
      try {
        sleep((int)(Math.random() * 100));
      } catch (InterruptedException e) { }
    }
  }
}
```

Clasa Consumator

```
class Consumator extends Thread {
  private Buffer buffer;

  public Consumator(Buffer b) {
    buffer = b;
  }

  public void run() {
    int value = 0;
    for (int i = 0; i < 10; i++) {
      value = buffer.get();
      System.out.println(
        "Consumatorul a primit:\t" + value);
    }
  }
}
```

Monitoare

Secţiune critica = segment de cod ce gestionează o resursă comună

Monitor = ”lacăt” asociat fiecărui obiect
Controlul accesului într-o secţiune critică se face prin cuvântul cheie **synchronized**

```
public synchronized void put(int number) {
  // buffer blocat de producator
  ...
  // buffer deblocat de producator
}
public synchronized int get() {
  // buffer blocat de consumator
  ...
  // buffer deblocat de consumator
}
```

Monitoare fine

```
class MonitoareFine {
  //Cele doua resurse ale obiectului
  Resursa x, y;
  //Folosim monitoarele a doua obiecte fictive
  Object xLacat = new Object(),
         yLacat = new Object();
  public void metoda() {
    synchronized(xLacat) {
      // Accesam resursa x
    }
    synchronized(yLacat) {
      // Accesam resursa y
    }
    synchronized(xLacat) {
      synchronized(yLacat) {
        // Accesam x si y
      }
    }
    synchronized(this) {
      // Accesam x si y
    }
  }
}
```

Semafoare

Metodele wait - notify

```
class Buffer {
  private int number = -1;
  private boolean available = false;

  public synchronized int get() {
    while (!available) {
      try {
        wait();
        // Asteapta producatorul sa puna o valoare
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    available = false;
    notifyAll();
    return number;
  }
```

```
  public synchronized void put(int number) {
    while (available) {
      try {
        wait();
        // Asteapta consumatorul sa preia valoarea
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    this.number = number;
    available = true;
    notifyAll();
  }
}
```

Probleme legate de sincronizare

Deadlock - "Problema filozofilor"

- Solicitarea resursele în aceeaşi ordine
- Monitoare care să controleze accesul la un grup de resurse
- Variabile care să informeze disponibilitatea resurselor fără a bloca monitoarele
- Arhitectura sistemului

Fire de execuţie inaccesibile

Operaţiile blocante IO nu trebuie făcute în metode sincronizate.

Variabile volatile

```
class TestVolatile {
  boolean test;
  public void metoda() {
    test = false;
    // *
    if (test) {
    // Aici se poate ajunge...
  }
}
```

Modificatorul `volatile` informează compilatorul să nu optimizeze codul în care aceasta apare, previzionând valoarea pe care variabila o are la un moment dat.

Gruparea firelor de execuţie

Clasa ThreadGroup

Fiecare fir de execuţie Java este membru al unui grup, afilierea fiind permanentă.

```
//Exemplu
ThreadGroup grup1 = new ThreadGroup("Producatori");
Thread p1 = new Thread(grup1, "Producator 1");
Thread p2 = new Thread(grup1, "Producator 2");

ThreadGroup grup2 = new ThreadGroup("Consumatori");
Thread c1 = new Thread(grup2, "Consumator 1");
Thread c2 = new Thread(grup2, "Consumator 2");
Thread c3 = new Thread(grup2, "Consumator 3");
```

Un grup poate avea ca părinte un alt grup - ierarhie de grupuri, cu rădăcina grupul implicit `main`

Variabile locale

Clasa ThreadLocal

Fiecare fir de execuţie poate gestiona variabile proprii ale căror valori sunt independente de celelalte fire de execuţie active.

```
//Exemplu
static ThreadLocal tlData = new ThreadLocal();

    public void aMethod() {

        // Retrieve value.
        Object obj = tlData.get();

        // Set value.
        tlData.set(obj);
    }
```

Comunicarea prin fluxuri de tip "pipe"

- **PipedReader, PipedWriter**
- **PipedOutputStream, PipedInputStream**

Conectarea fluxurilor

```
PipedWriter pw1 = new PipedWriter();
PipedReader pr1 = new PipedReader(pw1);
// sau
PipedReader pr2 = new PipedReader();
PipedWriter pw2 = new PipedWriter(pr2);
// sau
PipedReader pr = new PipedReader();
PipedWriter pw = new PipedWirter();
pr.connect(pw) //echivalent cu
pw.connect(pr);
```

```
class Producator extends Thread {
  private DataOutputStream out;
  public void run() {
    ...
    out.writeInt(i);
  }
}
class Consumator extends Thread {
  private DataInputStream in;
  public void run() {
    ...
    value = in.readInt();
  }
}
...
PipedOutputStream pipeOut = new PipedOutputStream();
PipedInputStream pipeIn = new PipedInputStream(pipeOut);
DataOutputStream out = new DataOutputStream(pipeOut);
DataInputStream in = new DataInputStream(pipeIn);
Producator p1 = new Producator(out);
Consumator c1 = new Consumator(in);
p1.start();
c1.start();
```

Clasele Timer şi TimerTask

Panificare unor acţiuni pentru o singură execuţie sau pentru execuţii repetate la intervale regulate.

- Crearea unei subclase `Actiune` a lui `TimerTask` şi supreadefinirea metodei `run`
- Crearea unui fir de execuţie prin instanţierea clasei `Timer`;
- Crearea unui obiect de tip `Actiune`;
- Planificarea la execuţie a obiectuluii de tip `Actiune`, folosind metoda `schedule` din clasa `Timer`;

Listing 2: Folosirea claselor `Timer` şi `TimerTask`

```
import java.util.*;
import java.awt.*;

class Atentie extends TimerTask {
  public void run() {
    Toolkit.getDefaultToolkit().beep();
    System.out.print(".");
  }
}

class Alarma extends TimerTask {
  public String mesaj;
  public Alarma(String mesaj) {
    this.mesaj = mesaj;
  }
  public void run() {
    System.out.println(mesaj);
  }
}

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

    // Setam o actiune repetitiva, cu rata fixa
    final Timer t1 = new Timer();
    t1.scheduleAtFixedRate(new Atentie(), 0, 1*1000);

    // Folosim o clasa anonima pentru o alta actiune
    Timer t2 = new Timer();
    t2.schedule(new TimerTask() {
      public void run() {
        System.out.println("S-au scurs 10 secunde.");
        // Oprim primul timer
        t1.cancel();
      }
    }, 10*1000);

    // Setam o actiune pentru ora 22:30
    Calendar calendar = Calendar.getInstance();
    calendar.set(Calendar.HOUR_OF_DAY, 22);
    calendar.set(Calendar.MINUTE, 30);
    calendar.set(Calendar.SECOND, 0);
    Date ora = calendar.getTime();
```

```
    Timer t3 = new Timer();
    t3.schedule(new Alarma("Toti copiii la culcare!"), ora);
  }
}
```

Desenarea şi firele de execuţie

Operaţiile consumatoare de timp vor fi făcute în fire de execuţie

Greşit

```
public void paint(Graphics g) {
  // Calcule complexe
  ...
  // Desenare
}
```

Corect

```
public void paint(Graphics g) {
  // Desenare
}
public void run() {
  //Calcule complexe
  repaint();
}
```

Fire de execuţie în Swing

SwingUtilities
invokeLater
invokeAndWait

```
Runnable doHello = new Runnable() {
  public void run() {
    System.out.println("Hello!");
  }
};
SwingUtilities.invokeLater(doHello);
```

java.awt.EventQueue