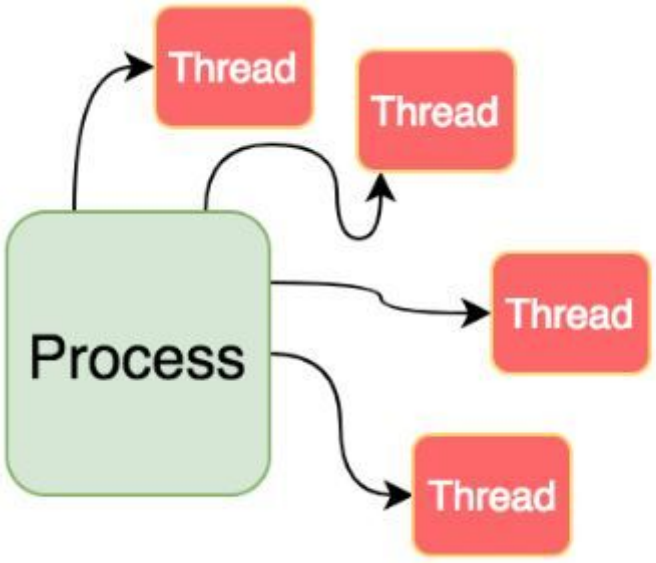
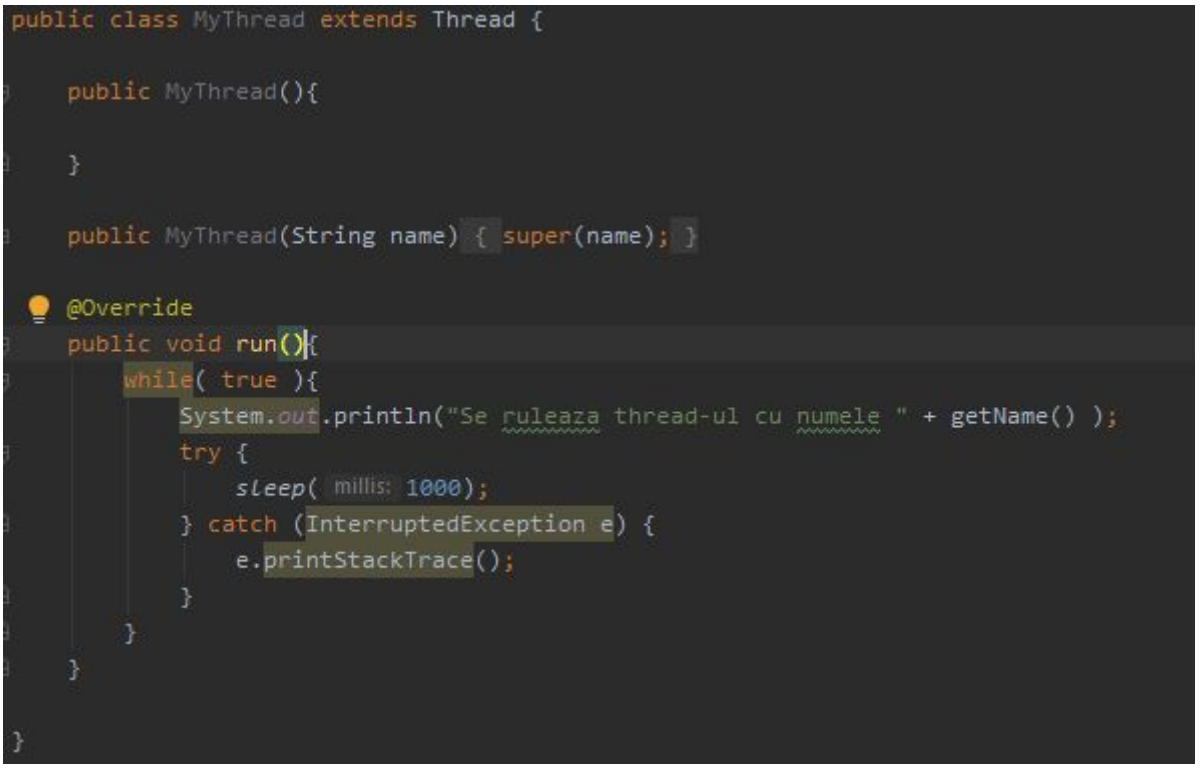
Laborator 9 - Fire de execuţie

- Un **fir de execuţie** este o succesiune secvenţială de instrucţiuni care se executa în cadrul unui **proces**(program în execuţie).
- Un fir de execuţie este unitatea de execuţie a unui proces.
- Un proces este format din mai multe fire de execuţie.
- Aplicaţiile care utilizează mai multe fire de execuţie pot executa în paralel mai multe sarcini.



Clasa **Thread**:

- o sa o gasiti în pachetul java.lang, şi conţine o serie de metode usefull pentru lucrul cu firele de execuţie.
- metoda principala este metoda **run()**, care trebuie sa conţină toate instrucţiunile pe care firul trebuie sa le execute.
- alte metode prezente în clasa Thread: start(), sleep() interrupt(), join(), getName(), setName(), destroy(), isInterrupted()



```java
public class MyThread extends Thread {

    public MyThread(){

    }

    public MyThread(String name) { super(name); }

    @Override
    public void run(){
        while( true ){
            System.out.println("Se ruleaza thread-ul cu numele " + getName() );
            try {
                sleep( millis: 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

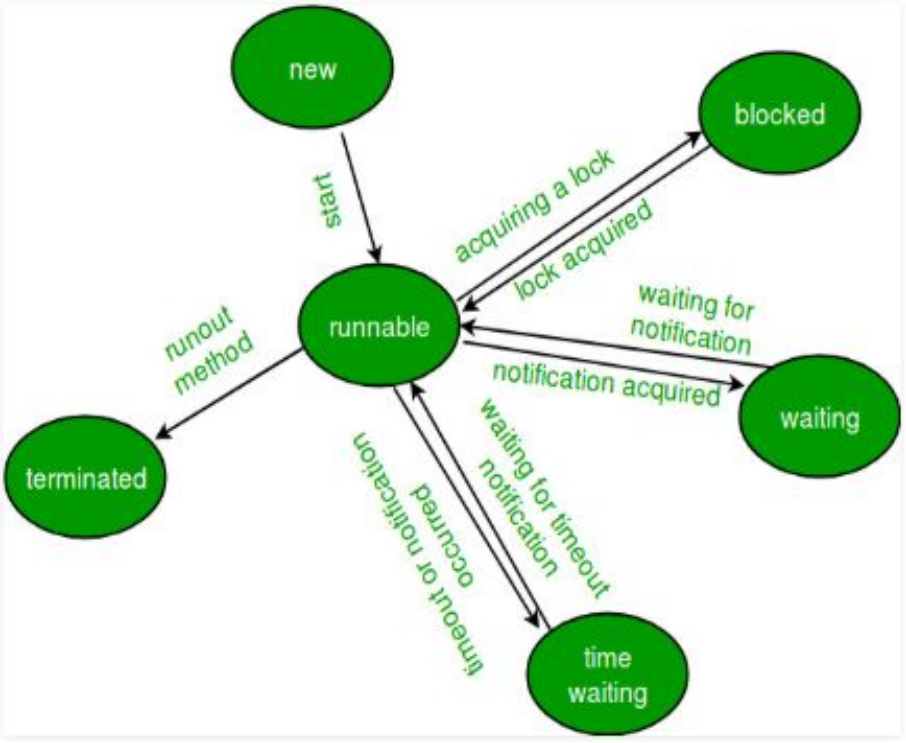
Interfaţa **Runnable**:

- a doua modalitate de a crea un fir de execuţie se utilizează cand clasa nou creata trebuie sa mostenească două clase fiindcă Java nu permite moştenire multiplă.
- interfaţa Runnable are o singura metoda , run(), care trebuie implementată
- clasele care implementează runnable nu au access la metodele clasei Thread(start(), stop()). Ca sa aiba access , vom crea un obiect de tip Thread care primeşte ca parametru obiectul care implementează Runnable.

```java
public class MyRunnableThread implements Runnable {

    @Override
    public void run() {
        while(true){
            System.out.println("Thread in executie");
            try {
                Thread.sleep( millis: 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Stările unui fir de execuţie



- **New**
- **Runnable**
- **Blocked**
- **Waiting**
- **Timed Waiting**
- **Terminated**

Prioritatea firelor de execuţie

- în Java sunt 10 niveluri de prioritate pentru thread-uri.
- avem şi trei constante definite în clasa Thread:

```
/**
 * The minimum priority that a thread can have.
 */
public static final int MIN_PRIORITY = 1;

/**
 * The default priority that is assigned to a thread.
 */
public static final int NORM_PRIORITY = 5;

/**
 * The maximum priority that a thread can have.
 */
public static final int MAX_PRIORITY = 10;
```

- JVM-ul utilizează proprietăţile firelor în planificarea firelor pentru execuţie.

```
public class Main {
    public static void main(String args[]) {
        SelfishThread s1, s2;

        s1 = new SelfishThread( name: "Firul 1");
        s1.setPriority (Thread.MAX_PRIORITY);

        s2 = new SelfishThread( name: "Firul 2");
        s2.setPriority (Thread.MIN_PRIORITY);

        s1.start();
        s2.start();
    }
}
```

- Prioritatea firului se seteaza cu metoda setPriority()

Thread-uri “Daemon”

Thread-urile daemon sunt nişte fire de execuţie speciale care au o prioritate redusă şi realizează anumite activităţi în background

- se distrug automat la terminarea celorlalte fire de execuţie
- un exemplu de fir de execuţie daemon: In Java, colectarea gunoiului se rulează pe un fir daemon
- cand toate user thread-uri sunt finalizate, daemon thread-urile o sa fie terminate automat
- poţi sa apelezi metoda setDaemon() înainte de apelul metodei start(); Odată ce firul de execuţie este pornit, nu mai poţi schimba daemon status-ul.
- isDaemon() ca sa verifici daca un fir de execuţie este de tip user sau tip daemon.
- JVM-ul nu o sa aştepte după niciun daemon thread să se finalizeze.

```
public class MyDaemonThread extends Thread{
    public MyDaemonThread(String name, boolean isDaemon){
        super(name);
        setDaemon(isDaemon);
    }

    @Override
    public void run(){
        for(int i=0;i<5;i++){
            try{
                sleep( millis: 2000 )  ;
            }
            catch( InterruptedException e ){}
            System.out.println(getName() + "se ruleaza");
        }
    }
}
```

Sincronizarea firelor de execuţie

- Conceptul de sincronizare ne permite sa limitam accesul firelor de execuţie la metode.

- Pentru ca o metoda sa fie sincronizata, trebuie sa adaugam în definirea metodei cuvantul cheie **`synchronized.`**

```
public synchronized void my_method() {...}
```

```
public class SynchronizedCounter {

    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Comunicarea între fire de execuţie:

- **wait()**
- **notify()**
- **notifyAll()**.