# Fire de executie (thread-uri)

- Program, proces, thread
- Programele au spatiu rezervat de adrese si executie
- Fiecare proces are propriul spatiu de adrese
- Thread-urile (firele de executie):
  - Impart acelasi spatiu de adrese;
  - Au acces la variabilele de clasa (comune);
  - Nu au acces la variabilele locale ale metodelor (care sunt asociate obiectelor);

- Fire de executie (thread-uri):
  - Clasa **Thread()**;
  - Mecanismul de fir de executie propriu zis (metode de control): **start(), sleep(), setPriority(), getPriority();**
  - Codul (metoda) executat(a) de firul de executie: pentru fiecare thread exista o metoda **run()**;

- Exista 2 metode de a creea un fir de executie:
  - Derivare din clasa **Thread()**; in acest caz trebuie sa se implementeze metoda run();
  - Implementarea interfetei **Runnable()**;

# Fire de executie (thread-uri)

SimpleThread.java | TestTwoThreads.java

```
public class SimpleThread extends Thread {
    public SimpleThread (String str) {
        super(str);
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(i + " " + getName());

            try {
             sleep((int)Math.random() * 1000);
            } catch(InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
```

```
DONE MAIN!
0 Paris
0 Bucharest
0 London
1 London
1 Paris
2 London
2 Paris
3 London
3 Paris
4 London
4 Paris
DONE! London
1 Bucharest
DONE! Paris
2 Bucharest
3 Bucharest
4 Bucharest
DONE! Bucharest
```

SimpleThread.java | TestTwoThreads.java

```
public class TestTwoThreads {

    public static void main(String args[]){
        new SimpleThread("Bucharest").start();
        new SimpleThread("London").start();
        new SimpleThread("Paris").start();

        System.out.println("DONE MAIN!");

    }
}
```

# Fire de executie (thread-uri)

- Prioritate (Thread.MIN_PRIORITY … Thread.MAX_PRIORITY);
- Un thread are initial o prioritate egala cu cea a thread-ului care l-a creat;
- OS cu divizarea timpului: fiecare thread se va executa o cuanta de timp; apoi este intrerupt si se executa un alt thread cu aceeasi prioritate (executie concurenta);
- OS fara divizarea timpului: un thread se executa pana la sfarsit (dupa care ii urmeaza un alt thread cu aceeasi prioritate) sau pana este intrerupt de un alt thread cu prioritate mai mare (executie secventiala);
- OS cu divizarea timpului :Thread-ul pierde procesorul:
  - La expirarea cuantei de timp curente;
  - Atunci cand executa un apel de metoda care conduce la blocare (**wait()**);
  - Cand executa un apel explicit de cedare (**yield()**);
- OS fara divizarea timpului :Thread-ul pierde procesorul:
  - Cand preda controlul in mod explicit (**yield()**);
  - Atunci cand executa un apel de metoda care conduce la blocare (**wait()**);
  - Cand se termina;
  - Cand exista un thread cu prioritate mai mare;
- Starile unui thread:
  - Creat (dupa initializarea cu **new()**);
  - Gata de executie (dupa apelarea metodei **start()**);
  - Suspendat (daca a fost apelata **sleep()** pentru el sau daca executa **wait()**);
  - Terminat (dupa termniarea metodei **run()** );

# Thread-uri: sincronizare: acces concurent la resurse

- Sincronizarea thread-urilor
  - Thread-uri: relatie de concurenta sau cooperare;
  - Concurenta: accesul la resursele comune trebuie sa se faca in mod coerent (un singur thread acceseaza la un moment dat resursa comuna); secventa de cod pentru care trebuie sa se asigure accesul exclusiv al unui singur thread: ***regiune critica***;
  - Cooperare: thread-urile trebuie sa schimbe informatii; procesele/firele de executie trebuie sa comunice atunci cand au ajuns la un anumit moment in executia lor; sincronizarea in acest caz presupune ca procesele/thread-urile trebuie sa se astepte (ele putand sa se execute cu viteze/prioritati diferite);
  - Sincronizarea prespune posibilitatea blocarii unui proces in asteptarea unui eveniment care depinde de alt proces;
  - Concurenta: evenimentul asteptat este “nici un alt proces nu se afla in regiunea critica”;
  - Cooperare: evenimentul asteptat este “procesele care comunica au ajuns in starea in care pot comunica”;

# Thread-uri: sincronizare: acces concurent la resursa comuna

Acces concurent **incorect** la o resursa comuna

NotSyncThread.java

```
public class NotSyncThread extends Thread {

    static int a = 0;
    static int b = 0;

    NotSyncThread(String name) {
        super(name);
    }


    void method() {
        System.out.println(getName() + "a = " + a + " b = " + b);
        a++;
        try {
            sleep((int) (Math.random() * 1000));
        }
        catch(InterruptedException e)
        {

        }
        b++;
    }

    public void run() {
        for (int i = 0; i < 9; i++) {
            method();
        }
        System.out.println(getName() + "DONE!");
    }
}
```

NotSyncThreadTest.java

```
public class NotSyncThreadTest extends Thread {

    public static void main (String args[]){
        new NotSyncThread("Thread 1: ").start();
        new NotSyncThread("Thread 2: ").start();
        new NotSyncThread("Thread 3: ").start();
        System.out.println("Main test DONE!");
    }
}
```

```
Thread 1: a = 0 b = 0
Thread 3: a = 0 b = 0
Thread 2: a = 0 b = 0
Main test DONE!
Thread 1: a = 3 b = 1
Thread 3: a = 4 b = 2
Thread 2: a = 5 b = 3
Thread 2: a = 6 b = 4
Thread 1: a = 7 b = 5
Thread 2: a = 8 b = 6
Thread 3: a = 9 b = 7
Thread 2: a = 10 b = 8
Thread 3: a = 11 b = 9
Thread 1: a = 12 b = 10
Thread 2: a = 13 b = 11
Thread 3: a = 14 b = 12
Thread 3: a = 15 b = 13
Thread 2: a = 16 b = 14
Thread 1: a = 17 b = 15
Thread 2: a = 18 b = 16
Thread 1: a = 19 b = 17
Thread 3: a = 20 b = 18
Thread 3: a = 21 b = 19
Thread 2: a = 22 b = 20
Thread 3: a = 23 b = 21
Thread 2: DONE!
Thread 1: a = 24 b = 23
Thread 1: a = 25 b = 24
Thread 3: DONE!
Thread 1: a = 26 b = 26
Thread 1: DONE!
```

# Thread-uri: sincronizare: acces concurent la metodele unui obiect

Acces concurent **incorect** la metodele unui obiect

CommonAccess.java

```
public class CommonAccess {

    int a = 0;
    int b = 0;

    void method(String name) {
        Thread t = Thread.currentThread();
        System.out.println(name + "a = " + a + " b = " + b);
        a++;
        try {
            t.sleep((int) (Math.random() * 1000));
        }
        catch(InterruptedException e)
        {
        }
        b++;
    }
}
```

NotSyncThread2Test.java

```
public class NotSyncThread2Test {
    public static void main (String args[]){
        CommonAccess a = new CommonAccess();
        new NotSyncThread2("Thread 1: ", a).start();
        new NotSyncThread2("Thread 2: ", a).start();
        new NotSyncThread2("Thread 3: ", a).start();
        System.out.println("Main test DONE!");
    }
}
```

NotSyncThread2Test.java | NotSyncThread2.java

```
public class NotSyncThread2 extends Thread {
    CommonAccess a;
    NotSyncThread2(String name, CommonAccess a) {
        super(name);
        this.a = a;
    }

    public void run() {
        for(int i = 0; i < 3; i++) {
            a.method(getName());
        }
        System.out.println("[THREAD] " + getName() + " DONE !");
    }
}
```

```
Main test DONE!
Thread 3: a = 0 b = 0
Thread 1: a = 0 b = 0
Thread 2: a = 2 b = 0
Thread 2: a = 3 b = 1
Thread 1: a = 4 b = 2
Thread 3: a = 5 b = 3
Thread 1: a = 6 b = 4
Thread 3: a = 7 b = 5
Thread 2: a = 8 b = 6
[THREAD] Thread 2:  DONE !
[THREAD] Thread 1:  DONE !
[THREAD] Thread 3:  DONE !
```

```
Thread 1: a = 0 b = 0
Thread 2: a = 0 b = 0
Thread 3: a = 0 b = 0
Main test DONE!
Thread 2: a = 3 b = 1
Thread 2: a = 4 b = 2
[THREAD] Thread 2:  DONE !
Thread 3: a = 5 b = 4
Thread 1: a = 6 b = 5
Thread 3: a = 7 b = 6
[THREAD] Thread 3:  DONE !
Thread 1: a = 8 b = 8
[THREAD] Thread 1:  DONE !
```

# Thread-uri: sincronizare: acces concurent la resursa comuna

SyncronizedThread.java | SyncronizedThreadTest.java

```
public class SyncronizedThread extends Thread {

    static int a = 0;
    static int b = 0;
    static Object obj = new Object();

    SyncronizedThread(String name){
        super(name);
    }
    void method() {
        System.out.println(getName() + "  a = " + a + " b = " + b);
        a++;
        try {
            sleep((int) (Math.random() * 1000));
        }
        catch(InterruptedException e)
        {

        }
        b++;
    }
    public void run() {
        for (int i = 0; i < 3; i++) {
            synchronized(obj) {
                method();
            }
        }
        System.out.println(getName() + "DONE!");
    }
}
```

SyncronizedThread.java | SyncronizedThreadTest.java

```
public class SyncronizedThreadTest {
    public static void main (String args[]){
        new SyncronizedThread("Thread 1: ").start();
        new SyncronizedThread("Thread 2: ").start();
        new SyncronizedThread("Thread 3: ").start();
        System.out.println("Main test DONE!");
    }
}
```

Acces concurent **corect** la o resursa comuna

Obiectul creat o este utilizat numai pentru a avea acces la zavorul sau; resursa comuna este formata din cele doua variabile si noul obiect o.

# Thread-uri: sincronizare: acces concurent la metodele unui obiect

CommonAccess.java | SynchronizedThread2.java

```
public class CommonAccess {

    int a = 0;
    int b = 0;

    synchronized void method(String name) {
        Thread t = Thread.currentThread();
        System.out.println(name + "a = " + a + " b = " + b);
        a++;
        try {
            t.sleep((int) (Math.random() * 1000));
        }
        catch(InterruptedException e)
        {
        }
        b++;
    }
}
```

CommonAccess.java | SynchronizedThread2.java

```
public class SynchronizedThread2 extends Thread {
    CommonAccess a;
    SynchronizedThread2(String name, CommonAccess a) {
        super(name);
        this.a = a;
    }

    public void run() {
        for(int i = 0; i < 3; i++) {
            a.method(getName());
        }
        System.out.println("[THREAD] " + getName() + " DONE !");
    }

}
```

## Acces concurent **corect** la metodele unui obiect

Metoda method are atributul synchronized
Astfel, executiile metodei din diferite thread-uri sunt seventiale, un singur thread poate sa fie in executia metodei respective la un moment dat; de aceasta data, zavorul (lock) este asociat obiectului de tip CommonAccess instantiat in main()

CommonAccess.java | SynchronizedThread2.java | SynchronizedThread2Test.java

```
public class SynchronizedThread2Test extends Thread   {
    public static void main (String args[]){
        CommonAccess a = new CommonAccess();
        new SynchronizedThread2("Thread 1: ", a).start();
        new SynchronizedThread2("Thread 2: ", a).start();
        new SynchronizedThread2("Thread 3: ", a).start();
        System.out.println("Main test DONE!");
    }
}
```

# Fire de executie (thread-uri)

## Sincronizarea pentru colaborare

# Thread-uri: sincronizare pentru colaborare

Exista un producator si 3 consumatori, producatorul face push intr-o stiva valorilor intre 1 si 10, consumatorii fac pop din aceeasi stiva. Accesul este sincronizat, obiectul folosit pentru sincronizare fiind chiar stiva

ProducerConsumer.java

```
import java.util.Stack;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * 1 producer and 3 consumers producing/consuming 10 items
 *
 * @author pt
 *
 */
public class ProducerConsumer {

    Stack<Integer> items = new Stack<Integer>();
    final static int NO_ITEMS = 10;

    class Producer implements Runnable {

        public void produce(int i) {
            System.out.println("Producing " + i);
            items.push(new Integer(i));
        }

        public void run() {
            int i = 0;
            // produce 10 items
            while (i++ < NO_ITEMS) {
                synchronized (items) {
                    produce(i);
                    items.notifyAll();
                }
                try {
                // sleep for some time,
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                }
                }
            }
    }
```

ProducerConsumer.java

```
    class Consumer implements Runnable {
        //consumed counter to allow the thread to stop
        AtomicInteger consumed = new AtomicInteger();

        public void consume() {
            if (!items.isEmpty()) {
                System.out.println("Consuming " + items.pop());
                    consumed.incrementAndGet();
            }
        }
        private boolean theEnd() {
            return consumed.get() >= NO_ITEMS;
        }
        public void run() {
            while (!theEnd()) {
                synchronized (items) {
                    while (items.isEmpty() && (!theEnd())) {
                    try {
                        items.wait(10);
                    } catch (InterruptedException e) {
                        Thread.interrupted();
                    }
                    }
                    consume();
                }
            }
        }
    public static void main(String args[]) {
        ProducerConsumer pc = new ProducerConsumer();
        Thread t1 = new Thread(pc.new Producer());
        Consumer consumer  = pc.new Consumer();
        Thread t2 = new Thread(consumer);
        Thread t3 = new Thread(consumer);
        Thread t4 = new Thread(consumer);
        t1.start();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }
        t2.start(); t3.start(); t4.start();
        try {
            t2.join(); t3.join(); t4.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

# Exemple de erori in utilizarea firelor de executie

# Thread-uri: sincronizare: solutie DCL (double check locking)

SomeClass1.java

```
public class SomeClass1 {

    static int globalCount = 0;
    private Helper helper = null;
    public Helper getHelper() {
        if(helper == null)
            helper = new Helper();
        return helper;
    }

    class Helper{
        private Helper() {
            System.out.println("Initializing Helper" + globalCount);
            globalCount ++;
        }
        void printMe() {
            System.out.println("Nothing to say");
        }
    }

    public static void main(String args[]) {
        SomeClass1 sC1 = new SomeClass1();
        SomeClass1 sC2 = new SomeClass1();

        Helper helper1 = sC1.getHelper();
        Helper helper2 = sC2.getHelper();
        helper1.printMe();
        helper2.printMe();
    }
}
```

SomeClass2.java

```
public class SomeClass2 {
    static int globalCount = 0;
    private Helper helper = null;
    // din pacate, aceasta functie este foarte costisitoare,
    // numai cateva threaduri fiind nevoie cu adevarat
    // sa fie sincronizate, dupa prima apelare a lui getHelper,
    // pentru tot restul care folosesc getHelper
    // avem un overhead foarte mare (de pana la 100 de ori
    // mai costisitoare decat fara "synchronized"
    public synchronized Helper getHelper() {
        if(helper == null)
            helper = new Helper();
        return helper;
    }

    class Helper{
        private Helper() {
            System.out.println("Initializing Helper" + globalCount);
            globalCount ++;
        }
        void printMe() {
            System.out.println("Nothing to say");
        }
    }
    public static void main(String args[]) {
        SomeClass2 sC1 = new SomeClass2();
        SomeClass2 sC2 = new SomeClass2();
        Helper helper1 = sC1.getHelper();
        Helper helper2 = sC2.getHelper();
        helper1.printMe();
        helper2.printMe();
    }
}
```

Utilizarea codului din exemplul (1) intr-un program cu mai multe thread-uri poate duce usor la erori, daca in mai multe thread-uri se testeaza daca obiectul de tip Helper este creat si il creeaza.
Exemplul (2) va duce la o executie corecta dar in schimb solutia este foarte costisitoare, numai primele thread-uri fiind nevoie sa fie sincronizate, pana la crearea obiectului, in rest, overhead-ul este prea mare.

# Thread-uri: sincronizare: solutie DCL (double check locking)

SomeClass3.java

```
public class SomeClass3 {

    static int globalCount = 0;
    private Helper helper = null;
    // se inlocuieste metoda sincronizata cu sincronizarea folosind obiectul curent (care este de tip Helper, si ar trebui sa fie
    // deci unic;
    // Sa presupunem insa urmatorul scenariu:
    // 1) Threadul 1 observa ca obiectul nu este initializat si incepe intializarea acestuia
    // 2) Codul generat de compilator permite actualizarea unei variabile partajate cu un obiect incomplet initializat
    // 3) Threadul 2 observa ca variabila partajata a fost initializata (aparent doar, de fapt) si nu pune stapanire pe
    // zavor (lock); cel mai probabil va avea loc un crash, pentru ca se acceseaza memoria din threadul 2, obiectul nefiind complet
    // initializat
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }

    class Helper{
        private Helper() {
            System.out.println("Initializing Helper" + globalCount);
            globalCount ++;
        }
        void printMe() {
            System.out.println("Nothing to say");
        }
    }

    public static void main(String args[]) {
        SomeClass3 sC1 = new SomeClass3();
        SomeClass3 sC2 = new SomeClass3();
        Helper helper1 = sC1.getHelper();
        Helper helper2 = sC2.getHelper();
        helper1.printMe();
        helper2.printMe();
    }
}
```

# Thread-uri: sincronizare: solutie DCL (double check locking)

SomeClass4.java

```
public class SomeClass4 {

    static int globalCount = 0;
    private volatile Helper helper = null;
    // Incepand cu Java 1.5, folosirea volatile va asigura ca nu se atribuie variabilei partajate
    // un obiect incomplet initializat
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }

    class Helper{
        private Helper() {
            System.out.println("Initializing Helper" + globalCount);
            globalCount ++;
        }
        void printMe() {
            System.out.println("Nothing to say");
        }
    }

    public static void main(String args[]) {
        SomeClass4 sC1 = new SomeClass4();
        SomeClass4 sC2 = new SomeClass4();

        Helper helper1 = sC1.getHelper();
        Helper helper2 = sC2.getHelper();
        helper1.printMe();
        helper2.printMe();
    }
}
```

# Thread-uri: sincronizare: solutie DCL (double check locking)

SomeClass5.java

```
public class SomeClass5 {

     static int globalCount = 0;
     // Solutia lui Bill Plugh (Mariland University - lazy initialization)
      private static class HelperHolder {
           public static Helper helper = new Helper();
       }

       public static Helper getHelper() {
           return HelperHolder.helper;
       }

   static class Helper{
       private Helper() {
           System.out.println("Initializing Helper" + globalCount);
           globalCount ++;
       }
       void printMe() {
           System.out.println("Nothing to say");
       }
   }

   public static void main(String args[]) {

       Helper helper1 = SomeClass5.getHelper();
       Helper helper2 = SomeClass5.getHelper();
       helper1.printMe();
       helper2.printMe();
   }
}
```

# Thread-uri: sincronizare:  Singleton (demo)

SingletonDemo.java

```
public class SingletonDemo {

        static int counter = 0;
        private static volatile SingletonDemo instance = null;

        private SingletonDemo() {
            System.out.println("SingletonDemo" + counter);
            counter++;
        }

        public static SingletonDemo getInstance() {
                if (instance == null) {
                        synchronized (SingletonDemo .class){
                                if (instance == null) {
                                        instance = new SingletonDemo ();
                                }
                        }
                }
                return instance;
        }

        public static void printMe() {
            System.out.println("SingletonDemo" + counter);
        }

        public static void main(String args[]) {
            SingletonDemo sD1 = new SingletonDemo();
            SingletonDemo sD2 = new SingletonDemo();
            SingletonDemo.printMe();
            SingletonDemo.printMe();
        }
}
```

# Thread-uri: sincronizare: Singleton (folosind singleton holder sau initializare “lenesa”)

LazyInitializationSingleton.java

```
public class LazyInitializationSingleton {
        static int counter = 0;
        // Private constructor prevents instantiation from other classes
        private LazyInitializationSingleton() { }

        /**
        * SingletonHolder is loaded on the first execution of Singleton.getInstance()
        * or the first access to SingletonHolder.INSTANCE, not before.
        */
        private static class SingletonHolder {
                public static final LazyInitializationSingleton INSTANCE = new LazyInitializationSingleton();
        }

        public static LazyInitializationSingleton getInstance() {
                System.out.println("LazyInitializationSingleton" + counter);
                counter++;
                return SingletonHolder.INSTANCE;
        }

        public static void printMe() {
            System.out.println("LazyInitializationSingleton" + counter);
        }

        public static void main(String args[]) {
            LazyInitializationSingleton.getInstance();
            LazyInitializationSingleton.getInstance();
            LazyInitializationSingleton.printMe();
            LazyInitializationSingleton.printMe();
        }
}
```