v 1.0-20181102

PROGRAMMAZIONE CONCORRENTE IN JAVA
- SECONDA PARTE -

Alessandro Ricci
a.ricci@unibo.it

SOMMARIO DEL MODULO

- Meccanismi di interazione e coordinazione di base
  - meccanismi di base di Java
    - synchronized, wait, notify, notifyAll
  - semafori e monitor in Java
  - esempi architetturali classici
    - produttori/consumatori, filosofi, readers-writers
  - supporti forniti a livello di libreria
    - collezioni concorrenti e synchronizers
    - stream con parallelizzazione
- Aspetti avanzati (cenni)
  - task-oriented programming e executors
  - programmazione asincrona
  - modelli ad attori

INTERAZIONE E COMUNICAZIONE

- Nel precedente modulo: introduzione alla programmazione concorrente in Java
  - in particolare: programmazione multi-threaded
  - metodologia task-oriented
- Qualsiasi programma (sistema) non banale => più thread che devono interagire/comunicare/cooperare per fornire le funzionalità dell'applicazione nel suo complesso
  - interazione e coordinazione come aspetti fondamentali nella progettazione di un sistema
- Quali modelli?

RICHIAMI DA S.O.: MODELLI DI INTERAZIONE

- Due modelli di riferimento
  - **a memoria comune**
    - c'è condivisione di memoria per i flussi di controllo nel sistema
    - comunicazione e interazione basata su accesso regolato a memoria condivisa e meccanismi di sincronizzazione
    - modello di riferimento per la programmazione multi-threaded
  - **a memoria locale**
    - non c'è condivisione di memoria
    - comunicazione basata su *scambio di messaggi*
      - è il modello di processi senza shared memory, che comunicano unicamente via segnali o scambio di messaggi
    - modello di riferimento per i sistemi distribuiti / di rete

TIPI DI INTERAZIONE NEL MODELLO A MEMORIA COMUNE

- **Cooperazione**
  - interazioni volute e necessarie per realizzare il corretto funzionamento dell'insieme dei processi (o thread) interagenti
    - **comunicazione**
      - scambio di informazioni
    - **sincronizzazione**
      - meccanismi per forzare un ordine temporale fra le azioni dei processi
- **Competizione**
  - interazioni non volute ma necessarie per realizzare il corretto funzionamento dell'insieme dei processi (o thread)
    - **mutua esclusione**
      - accesso mutuamente esclusivo a risorse
    - **sezioni critiche**
      - esecuzione mutuamente esclusiva di blocchi di azioni da parte dei processi
- **Interferenze**
  - interazioni non volute e non necessarie ai fini del corretto funzionamento del sistema, anzi tipicamente dannose
    - corse critiche

SINCRONIZZAZIONE

- Meccanismi e tecniche per realizzare **un ordine temporale** fra le azioni dei processi
  - ad esempio
    - fare in modo che l'azione o attività di un processo possa essere eseguita solo dopo che l'azione o attività di un altro processo si sia conclusa
  - la comunicazione può essere usata come meccanismo per avere sincronizzazione
- Forme di sincronizzazione
  - diretta o esplicita
    - sincronizzazione in caso di cooperazione
  - indiretta o implicita
    - sincronizzazione in caso di competizione

SINCRONIZZAZIONE IN JAVA: MECCANISMI DI BASE

- Sincronizzazione implicita e mutua esclusione
  - blocchi e metodi synchronized
  - realizzare classi thread-safe
- Sincronizzazione esplicita e coordinazione
  - primitive wait, notify, notifyAll

JAVA MEMORY MODEL

- Il modello di memoria adottato in Java fornisce un insieme molto ristretto di garanzie in merito alla semantica dell'accesso concorrente in lettura/scrittura o solo in scrittura a variabili (e.g. campi di un oggetto) condivise
  - l'accesso puramente in lettura non crea ovviamente problemi
- In particolare:
  - accesso a campi di tipo boolean, char, int e a campi che contengono il riferimento ad un oggetto è garantito essere atomico
  - accesso a campi di tipo long, double non è garantito essere atomico
- Nel prosieguo di questo modulo si considerano tuttavia idiomi e pattern che evitano il più possibile l'accesso concorrente R/W ai medesimi campi

METODI/BLOCCHI synchronized

- Dichiarando un metodo synchronized si vincola l'esecuzione del metodo ad un solo thread per volta.
- I thread che ne richiedono l'esecuzione mentre già uno sta eseguendo vengono automaticamente sospesi dalla JVM
  - in attesa che il thread in esecuzione esca dal metodo
- Dichiarando più metodi synchronized il vincolo viene esteso a tutti i metodi in questione
  - se un thread sta eseguendo un metodo synchronized, ogni thread che richiede l'esecuzione di un qualsiasi altro metodo synchronized viene sospeso e messo in attesa.
  - i metodi synchronized sono dunque mutuamente esclusivi
- Tale vincolo non vale nei confronti dei metodi non synchronized
  - il fatto che un thread sta eseguendo un metodo synchronized, non vieta ad altri thread di eseguire concorrentemente eventuali metodi non synchronized dell'oggetto stesso

ESEMPIO #1

```
package oop.concur.part2;

public class UserA extends Thread {
    private A obj;
    public UserA(A obj){
      this.obj = obj;
    }
    public void run(){
      log("before invoking m");
      obj.m();
      log("after invoking m");
    }
    private void log(String msg){
      System.out.println("["+Thread.currentThread()+"] "+msg);
    }
}
```

```
package oop.concur.part2;
public class A {
  public synchronized void m(){
    System.out.println("Thread "+Thread.currentThread()+" entered.");
    try {
      Thread.sleep(5000);
    } catch (Exception ex){}
    System.out.println("Thread "+Thread.currentThread()+" exited.");
  }
}
```

TEST

- Nel test creiamo due thread (di classe UserA) accedono concorrentemente ad un oggetto condiviso di classe A, invocando il metodo synchonized

```
package oop.concur.part2;
public class TestUserA {
    public static void main(String[] args) {
        A obj = new A();
        UserA userA = new UserA(obj);
        UserA userB = new UserA(obj);
        userA.start();
        try {
            Thread.sleep(500);
        } catch (Exception ex){
        }
        userB.start();
    }
}
```

Output:

```
[Thread[Thread-0,5,main]] before invoking m
Thread Thread[Thread-0,5,main] entered.
[Thread[Thread-1,5,main]] before invoking m
Thread Thread[Thread-0,5,main] exited.
[Thread[Thread-0,5,main]] after invoking m
Thread Thread[Thread-1,5,main] entered.
Thread Thread[Thread-1,5,main] exited.
[Thread[Thread-1,5,main]] after invoking m
```

- Eseguendo il test è possibile verificare come che userB riesce ad entrare nel metodo m solo quando userA è uscito

MUTUA ESCLUSIONE E MONITOR

- E' possibile sfruttare il meccanismo synchronized per realizzare la forma di mutua esclusione propria dei monitor (vedi S.O.)
- E' sufficiente dichiarare tutti i metodi pubblici synchronized
- Esempio contatore come monitor:

```
package oop.concur.part2;

public class Counter {
  private int cont;
  public Counter (){ cont = 0; }
  public synchronized void inc(){
    cont++;
  }
  public synchronized void dec(){
    cont--;
  }
  public synchronized int getValue(){
    return cont;
  }
}
```

TEST DEL CONTATORE

```
package oop.concur.part2;

public class CounterUserA extends Thread {
    private int ntimes;
    private Counter counter;
    public CounterUserA(CounterSync c, int n){
        ntimes=n;
        counter = c;
    }
    public void run(){
        log("starting - counter value is "+counter.getValue());
        for (int i=0; i<ntimes; i++){
            counter.inc();
        }
        log("completed - counter value is "+counter.getValue());
    }
    private void log(String msg){
        System.out.println("[COUNTER USER A] "+msg);
    }

}
```

TEST DEL CONTATORE

```
package oop.concur.part2;

public class CounterUserB extends Thread {
    private int ntimes;
    private Counter counter;
    public CounterUserB(CounterSync c, int n){
        ntimes=n;
        counter = c;
    }
    public void run(){
        log("starting - counter value is "+counter.getValue());
        for (int i=0; i<ntimes; i++){
            counter.dec();
        }
        log("completed - counter value is "+counter.getValue());
    }
    private void log(String msg){
        System.out.println("[COUNTER USER B] "+msg);
    }
}
```

TEST DEL CONTATORE

- Lancio di due thread...

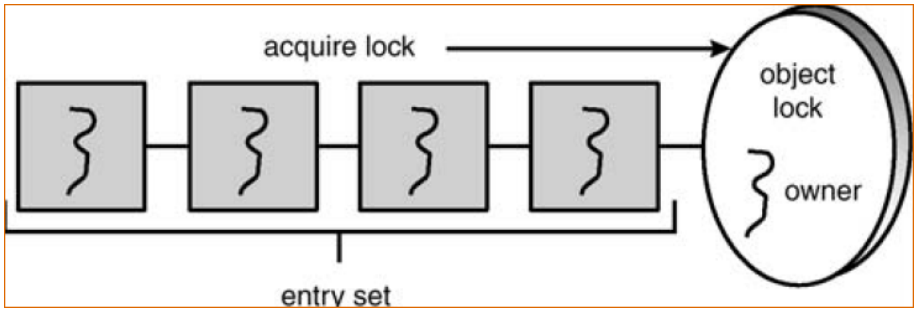
```
package oop.concur.part2;
public class TestConcurrentCounter {
    public static void main(String[] args) throws Exception {
        Counter c = new Counter();
        int ntimes = Integer.parseInt(args[0]);
        CounterUserA2 agentA = new CounterUserA2(c,ntimes);
        CounterUserB2 agentB = new CounterUserB2(c,ntimes);
        agentA.start(); agentB.start();
        agentA.join(); agentB.join();
        System.out.println("Counter final value: "+c.getValue());
    }
}
```

L'utilizzo di synchronized fa in modo che non ci siano interferenze e il conteggio risultante alla fine è sempre 0

```
[COUNTER USER A] starting - counter value is 0
[COUNTER USER B] starting - counter value is 723829
[COUNTER USER A] completed - counter value is 210926
[COUNTER USER B] completed - counter value is 0
Counter final value: 0
```

METODI SYNCHRONIZED: FUNZIONAMENTO

- A livello di implementazione il meccanismo di coordinazione synchronized è realizzato mediante un lock associato nativamente ad ogni oggetto.
  - l'esecuzione di un metodo synchronized comporta prima l'acquisizione del lock:
    - se il lock è già posseduto da un altro thread, il richiedente è bloccato / sospeso e inserito nella lista dei thread in attesa, chiamata entry set del lock.
    - se il lock è disponibile invece, il thread diviene proprietario del lock dell'oggetto ed esegue il metodo.
  - quando il proprietario rilascia il lock - o perché ha terminato l'esecuzione del metodo o perché deve esser sospeso, in seguito all'esecuzione di una wait - se l'entry set non è vuota, viene rimosso uno thread e ad esso viene assegnato il lock, ripristinandone l'esecuzione.
- La JVM non specifica alcuna politica di gestione dell'entry set
  - tipicamente viene utilizzata una politica FIFO



BLOCCHI synchronized

- Il meccanismo synchronized può essere utilizzato anche a livello di blocchi di codice, con una granularità quindi più fine rispetto al caso dei metodi

```
AnyClass myLockObject;
...
synchronized (myLockObject){
  <statements>
}
```

- Semantica del costrutto
  - prima di eseguire le istruzioni specificate all'interno dello blocco (statements), viene acquisito il lock sull'oggetto specificato
  - nel caso in cui il lock sia già stato acquisito, il thread corrente viene sospeso nell'entry set dell'oggetto specificato (myLockObject)
  - al termine dell'esecuzione delle istruzioni, in uscita dal blocco synchronized, viene automaticamente rilasciato il lock

SEZIONI CRITICHE CON BLOCCHI SYNCHRONIZED: ESEMPIO

- Nell'esempio, due thread utilizzano un blocco synchronized su un oggetto condiviso per realizzare due sezione critiche

```
class MyWorkerA extends Thread {
  private Object lock;
  public MyWorkerA(Object lock){
    this.lock = lock;
  }
  public void run(){
    while (true){
      System.out.println("a1");
      synchronized(lock){
        System.out.println("a2");
        System.out.println("a3");
      }
    }
  }
}
```

```
class MyWorkerB extends Thread {
  private Object lock;
  public MyWorkerB(Object lock){
    this.lock = lock;
  }
  public void run(){
    while (true){
      synchronized(lock){
        System.out.println("b1");
        System.out.println("b2");
      }
      System.out.println("b3");
    }
  }
}
```

```
public class TestCS {
  public static void main(String[] args) {
    Object lock = new Object();
    new MyWorkerA(lock).start();
    new MyWorkerB(lock).start();
  }
}
```

SINCRONIZZAZIONE ESPLICITA: wait, notify, notifyAll

- *wait, notify, notifyAll*
  - metodi pubblici definiti nella root class java.lang.Object
    - quindi ereditati da qualsiasi nuova classe che scriviamo
- Semantica
  - **wait**
    - l'invocazione della wait provoca la sospensione del thread corrente (con rilascio del lock sull'oggetto), fino a quando un altro thread non esegua una notify o notifyAll sul medesimo oggetto
    - i thread sospesi vengono incluso in un insieme chiamato wait set
  - **notify**
    - provoca il risveglio di uno degli eventuali thread sospesi sul medesimo oggetto con la wait
  - **notifyAll**
    - provoca il risveglio di tutti i thread sospesi con la wait
- **NOTA IMPORTANTE: Per poter chiamare uno qualsiasi di questi metodi è necessario avere prima ottenuto il lock sull'oggetto**
  - tipicamente vengono chiamati all'interno di metodi synchronized

ESEMPIO #2: UNA CELLA DI MEMORIA SINCRONIZZATA

- NOTA
  - notifyAll anziché notify perché potrebbero esserci più thread che hanno eseguito la wait e sono sospesi...

```
class MySynchCell {

  private Info info;
  private boolean ready = false;

  public synchronized Info get(){
    while (!ready){
      wait();
    }
    return info;
  }

  public synchronized void set(Info info){
    ready = true;
    this.info = info;
    notify();
  }
}
```

Il metodo get blocca il thread chiamante se il valore da leggere non è ancora stato settato

NOTA:
- perché è necessario usare il flag ready?
- l'implementazione è corretta in caso di più thread che eseguono la get?

ESEMPIO #2b: UNA CELLA DI MEMORIA SINCRONIZZATA

- NOTA
  - notifyAll anziché notify perché potrebbero esserci più thread che hanno eseguito la wait e sono sospesi...

```
class MySynchCell {

  private Info info;
  private boolean ready = false;

  public synchronized Info get(){
    while (!ready){
      wait();
    }
    return info;
  }

  public synchronized void set(Info info){
    ready = true;
    notifyAll();
  }
}
```

- versione funzionante anche in caso di più thread getter

IMPLEMENTAZIONE DI UN SEMAFORO

```
public class Sem {
  private int c; // c >= 0

  public Sem(int c){
    if (c < 0){
      throw new IllegalArgumentException();
    }
    this.c = c;
  }

  public synchronized void await() throws InterruptedException {
    while (c == 0){
      wait();
    }
    c--;
  }

  public synchronized void signal(){
    c++;
    notify();
  }
}
```

C'è già una implementazione del costrutto semaforico nella libreria java.util.concurrent: classe Semaphore - metodi acquire (await) e release (signal)

ESEMPIO USO SEMAFORI: MUTUA ESCLUSIONE

```
class MyAgentA extends Thread {
  private Sem mutex;
  public MyAgentA(Sem mutex){
    this.mutex = mutex;
  }
  public void run(){
    while (true){
      System.out.println("a1");
      mutex.await();
      System.out.println("a2");
      System.out.println("a3");
      mutex.signal();
    }
  }
}
```

```
class MyAgentB extends Thread {
  private Sem mutex;
  public MyAgentB(Sem mutex){
    this.mutex = mutex;
  }
  public void run(){
    while (true){
      System.out.println("b1");
      mutex.await();
      System.out.println("b2");
      System.out.println("b3");
      mutex.signal();
    }
  }
}
```

```
public class TestCSwithSem {
  public static void main(String[] args) {
    Sem mutex= new Sem(1);
    new MyWorkerA(mutex).start();
    new MyWorkerB(mutex).start();
  }
}
```

ESEMPIO USO SEMAFORI: SINCRONIZZAZIONE AZIONI

```
class AgentA extends Thread {
  private Sem synch;
  public AgentA(Sem synch){
    this.synch = synch;
  }
  public void run(){
    System.out.println("a");
    sleepAbit();
    synch.signal();
  }
  private void sleepAbit(){
    try { Thread.sleep(2000); }
    catch (Exception ex){}
  }
}
```

```
class AgentB extends Thread {
  private Sem synch;
  public AgentB(Sem synch){
    this.synch = synch;
  }
  public void run(){
    synch.await();
    System.out.println("b");
  }
}
```

```
public class TestSemSynch {
  public static void main(String[] args) {
    Sem s = new Sem(0);
    new AgentA(s).start();
    new AgentB(s).start();
  }
}
```

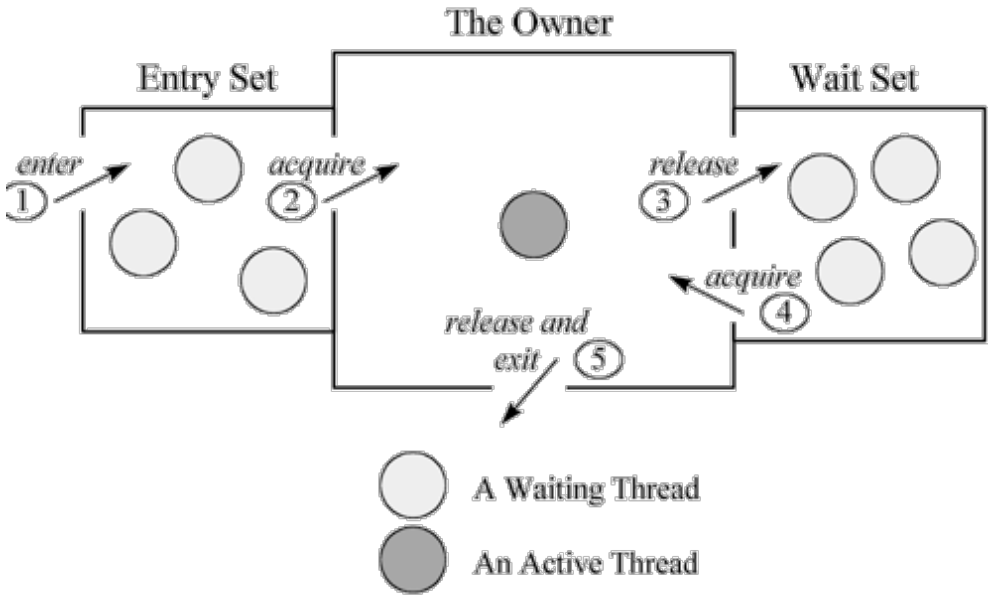
ESEMPIO USO SEMAFORI LIB JAVA: java.util.concurrent.Semaphore

```
class MyAgentA extends Thread {
  private Semaphore mutex;
  public MyAgentA(Sem mutex){
    this.mutex = mutex;
  }
  public void run(){
    while (true){
      System.out.println("a1");
      try {
        mutex.acquire();
        System.out.println("a2");
        System.out.println("a3");
        mutex.release();
      } catch (InterruptedException ex( {}
    }
  }
}
```

```
public class TestCSwithSem {
  public static void main(String[] args) {
    Semaphore mutex= new Semaphore(1);
    new MyWorkerA(mutex).start();
    new MyWorkerB(mutex).start();
  }
}
```

FUNZIONAMENTO: ENTRY SET E WAIT SET

- Step
  - invocazione metodo synchronized
  - si ottiene il lock
  - esecuzione di una wait, con rilascio del lock
  - notifica da parte del thread attivo e ottenimento del lock
  - uscita dal metodo synchronized



MONITOR IN JAVA - IN SINTESI

- Con i meccanismi di base in Java è possibile sviluppare forme semplificate di monitor, con una sola condition variable
- Punti importanti
  - tutti i metodi pubblici devono essere dichiarati synchronized
  - i metodi wait e notify/notifyAll sono equivalenti alle primitive awaitC e signalC/signalAllC sulla variabile condizione
  - è opportuno usare sempre il costrutto while quando si testano le condizioni sulla condition variable
    - fenomeno delle spourious wake-up
    - semantica di segnalazione, variante della Signal & Continue (E = W < S)

MONITOR IN JAVA: ASPETTI AVANZATI

- La semantica della notify/notifyAll è una variante della semantica Signal & Continue
  - E = W < S
    - ovvero nel riottenere il lock, chi esegue la wait compete con chi è nell'entry set
- La disponibilità di wait e notify/notifyAll equivale ad avere una sola condition variable utilizzabile nel monitor (oggetto)
  - quindi una sola variabile viene utilizzata per sospendere/segnalare
  - questo implica la necessità - nel caso di condizioni di sospensione diverse, a meno di casi particolari - di utilizzare notifyAll per svegliare tutti i thread in attesa
    - fra cui anche quello che effettivamente vogliamo svegliare)
  - ...e dell'uso del while per ritestare le condizioni di sospensione
    - e verificare di esser stati svegliati perché effettivamente valgono le condizioni per proseguire
- E' possibile realizzare monitor con più variabili condizioni indirettamente, combinando in modo opportuno i meccanismi
  - classi ReentrantLock e Condition nella libreria java.util.concurrent

ESEMPI ARCHITETTURALI

- **Produttori-Consumatori**
  - bounded buffer monitor
- **Dining Philosophers**
  - semafori per mutua esclusione sulle forchette + strategia per evitare deadlock
- **Lettori-Scrittori**
  - monitor che funge da distributore di lock

LIBRERIA java.util.concurrent

- La libreria java.util.concurrent include un ricco insieme di costrutti di alto livello per semplificare lo sviluppo di applicazioni concorrenti, tra cui:
  - **Concurrent Collections**
    - implementazione efficiente e thread-safe di strutture dati come liste, mappe, code, stack da utilizzare in contesti concorrenti
  - **Synchronizers**
    - implementazione di meccanismi e costrutti classici per la coordinazione di thread (semaphores, latches, barriers,....)

CONCURRENT COLLECTIONS

- Implementazione delle collections di Java appositamente pensata per essere efficace nel caso di accessi concorrenti da più thread
  - thread-safe
  - ottimizzata in modo da minimizzare le parti sequenziali
- Fra le classi principali:
  - **ConcurrentHashMap**
    - versione concorrente delle hash map
  - **Queue and BlockingQueue**
    - interfacce che rappresentano code, con diverse implementazioni
    - alcune sono utili per implementare direttamente dei bounded-buffer
  - **CopyOnWriteArrayList**
    - versione concorrente di ArrayList

SYNCHRONIZERS

- Nel terminologia adottata dalla libreria, un synchronizer è un qualsiasi oggetto passivo che serve per coordinare il flusso di controllo dei thread che vi interagiscono
- Sono i componenti di base che incapsulano funzionalità di coordinazione classiche, utili in tutte le applicazioni
- Tipi principali inclusi nella libreria:
  - **Locks**
  - **Semaphores**
  - **Latches**
  - **Barriers**
- Proprietà generali synchronizer:
  - incapsulano uno stato che determina quando il thread che li invoca può proseguire oppure essere bloccato
  - forniscono metodi per manipolare tale stato
  - forniscono metodi che permettono di attendere in modo efficiente il verificarsi di tale stato - eventualmente bloccando il flusso di controllo del thread chiamante

PARALLEL STREAMS

- Eseguono le operazioni viste sugli stream sfruttando parallelismo
  - partizionamento automatico di uno stream in più sub-stream
  - le operazioni di aggregazione iterano sui sub-stream in parallelo e poi aggregano

```
double average = roster
    .parallelStream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

- Nota importante:
  - l'esecuzione che sfrutta il parallelismo non implica necessariamente che le performance siano migliori del caso sequenziale
    - causa overhead creazione e gestione thread

ASPETTI AVANZATI - CENNI

- Programmazione **task-oriented**
  - executors
- Programmazione **asincrona**
  - modelli ad eventi / event-loop
  - programmazione reattiva (RxJava)
- Programmazione ad **attori**
  - oggetti + concorrenza
  - scambio asincrono di messaggi

TASK ED EXECUTORS

- Supporto per decomposizione di un problema in **task**
  - unità logica di lavoro, indipendente dalle altre
  - diverso dal concetto di thread
    - un task è eseguito/mappato su un thread
    - un thread può eseguire uno o giù task
- Framework Executors
  - fornisce un supporto diretto per disaccoppiare la fase di invio/ submit di un task da compiere e la sua esecuzione
- Un **Executor:**
  - accetta task da eseguire
    - in modo *asincrono* rispetto al flusso di controllo di chi lo invia
  - manda in esecuzione i task tipicamente in parallelo usando un pool di thread
  - si possono creare tipi diversi di executor che usano strategie diverse per l'esecuzione dei task
- Pattern **master-worker**

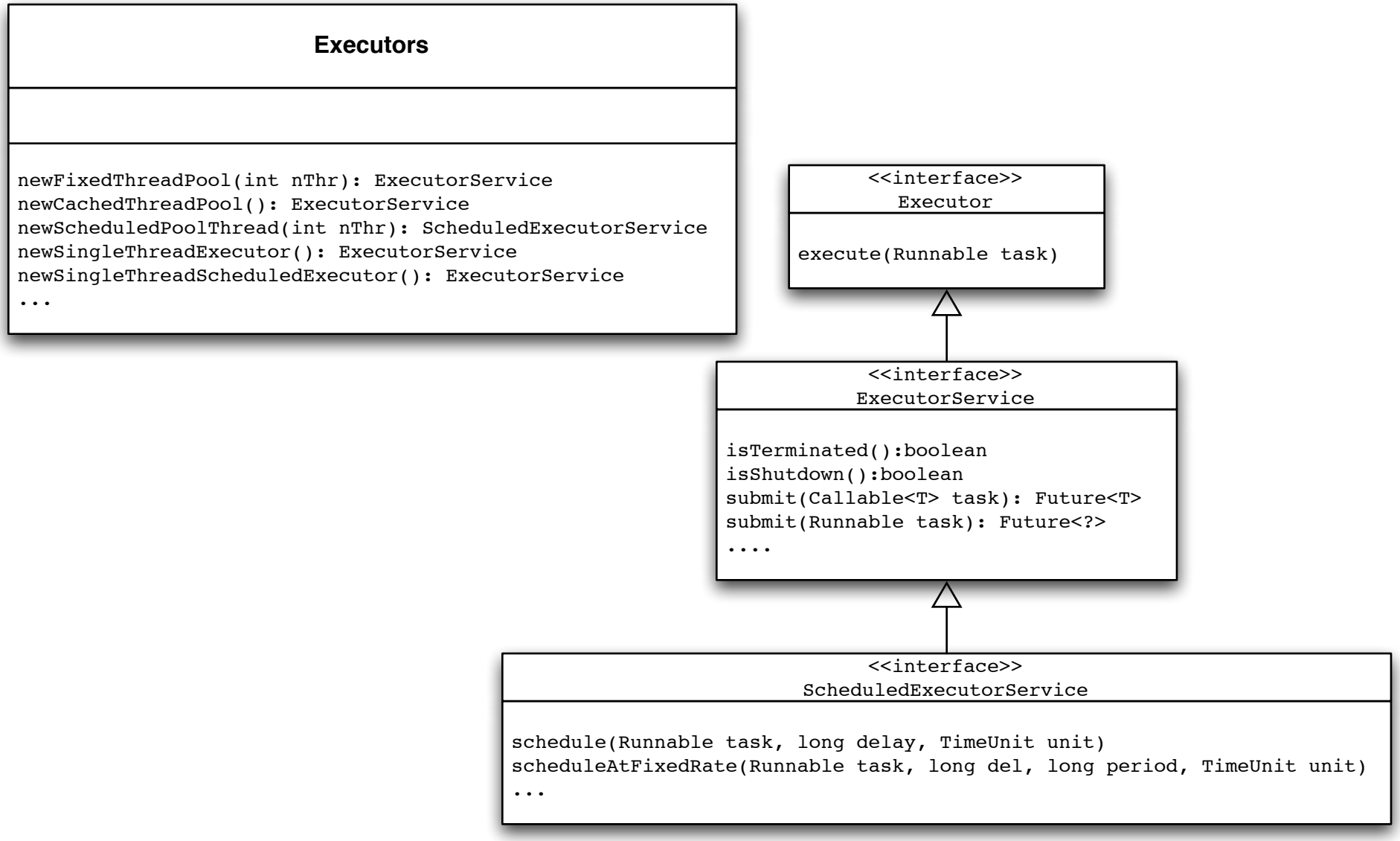
ESEMPIO SIMPLE SORT CON TASK

```
private static void sortPar(int[] data){
    int[] v = Arrays.copyOf(data, data.length);
    int middle = v.length / 2;

    int threadPoolSize = Runtime.getRuntime().availableProcessors() + 1;
    ExecutorService exec = Executors.newFixedThreadPool(threadPoolSize);

   exec.execute(() -> {
        Arrays.sort(v, 0, middle);
    });
    exec.execute(() -> {
        Arrays.sort(v, middle, v.length);
    });

    exec.shutdown();
    try {
        exec.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
        merge(v,data,0,middle,middle,v.length);
    } catch (InterruptedException ex){
        ex.printStackTrace();
    }
}
```

EXECUTOR API

- Interfaccia Executor
  - metodo execute per fare il submit dei task
  - le classi che implementano questa interfaccia incapsulano l'execution policy dei compiti
- Task descritti dall'interfaccia Runnable
  - metodo run usato per definire il comportamento dei task
- classe Executors
  - factory che consente di creare istanze concrete di executor che implementano una specifica execution policy
  - classe di utilità che fornisce funzionalità per raccogliere statistiche ed in generale gestire e monitorare l'applicazione

FRAMEWORK AND API



EXECUTORS FACTORY: EXECUTORS DISPONIBILI

- **FixedThreadPool**
  - usa un pool di thread con dimensione prefissata per eseguire i task inviati
  - task eseguiti in ordine di inserimento (FIFO)
- **CachedThreadPool**
  - crea nuovi thread se necessario e riduce la dimensione del pool nel caso sia superiore alla domanda
- **SingleThreadExecutor**
  - usa un singolo thread worker, rimpiazzandolo in caso di failure
- **ScheduledThreadPool**
  - come fixed thread pool, con supporto per task periodici o ritardati nel tempo

PROGRAMMAZIONE ASINCRONA

- Programmazione asincrona, in generale:
  - modelli, tecniche, architetture che permettono di mandate in esecuzione computazioni che vengono svolte in modo *asincrono*, **non bloccante** rispetto a chi le ha richieste/generate di gestirne il risultato, astraendo dalla gestione dei thread sottostante
- Due macro-categorie
  - basata su Task
  - basata su eventi

PROGRAMMAZIONE ASINCRONA BASATA SU TASK

- Nel framework Executor, i task vengono eseguiti in modo asincrono rispetto al flusso di controllo che li invia

```
…
executor.exec(()->{ do_this(); })
executor.exec(()->{ do_that(); })
…
```

- Per recuperare i risultati: uso di ***future***

```
interface ArchiveSearcher { String search(String target); }
 class App {
   ExecutorService executor = ...
   ArchiveSearcher searcher = ...
   void showSearch(final String target) throws InterruptedException {
     Future<String> future
       = executor.submit(() -> { return searcher.search(target); }});
     displayOtherThings(); // do other things while searching
     displayText(future.get()); // use future
     …
  }
 }
```

PROGRAMMAZIONE ASINCRONA BASATA SU EVENTI

- I framework che supportano questo modello di programmazione permettono di specificare una **callback** come *continuazione* che deve essere eseguita *quando* il risultato della computazione asincrona è disponibile
- Esempio astratto:

```
…
actionA();
call asyncTask (<params>, (result) -> {
  actionC();
});
actionB();
…
```

  La callback è un handler associato all'evento relativo al completamento della computazione asincrona
- Modello di riferimento usato nella programmazione web, mobile
- Esempi di framework
  - Vert.x (Java), node.js (Javascript)

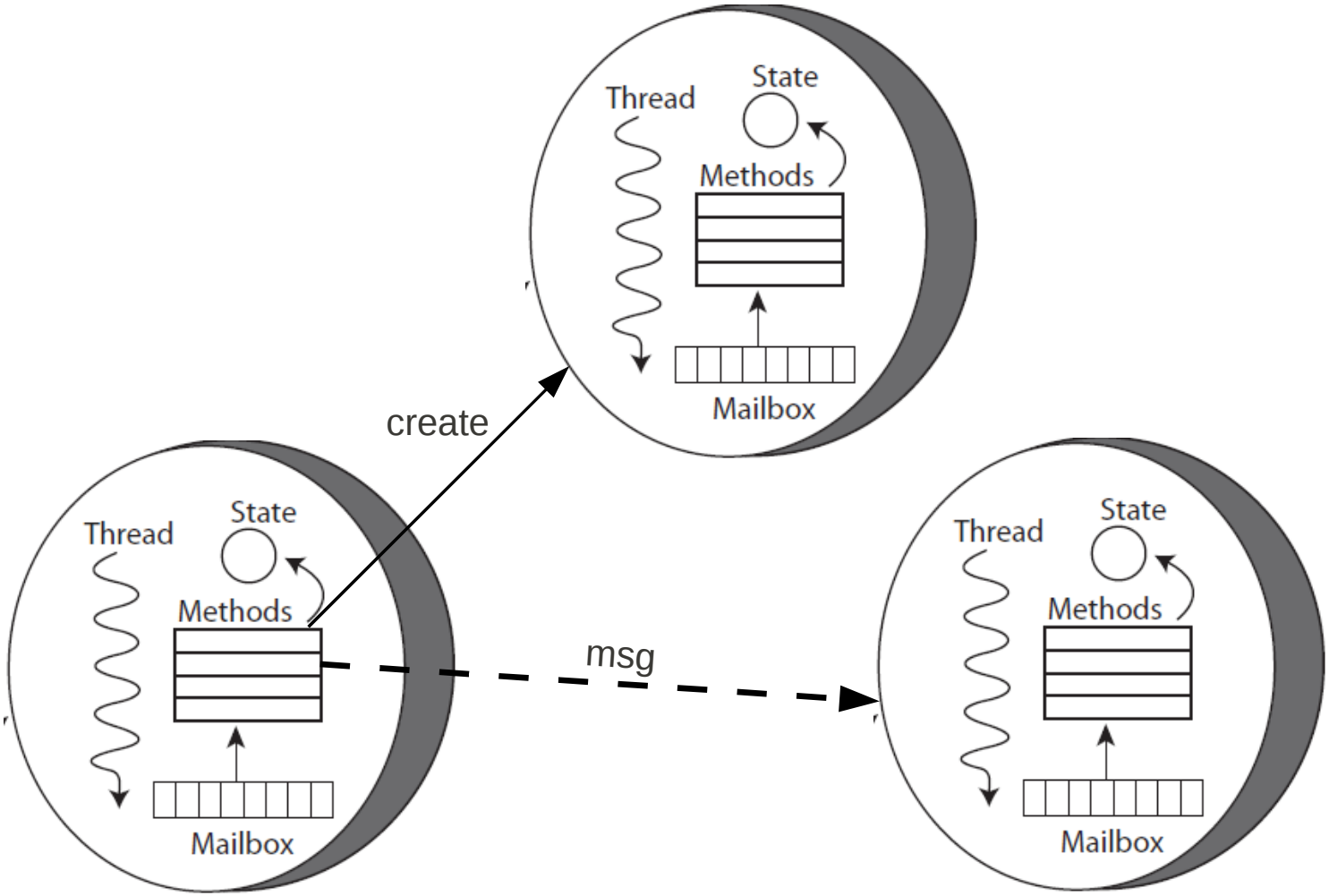
PROGRAMMAZIONE AD ATTORI

- Punto chiave del modello ad attori: interazione esclusivamente basata su **scambio asincrono di messaggi**
  - no memoria condivisa, no operazioni bloccanti
- Concetto/astrazione di **attore**
  - entità attiva o autonoma
    - *dotata di un proprio flusso di controllo logico*
  - reattiva
    - computa quando riceve un messaggio, mandando in esecuzione l'handler associato
    - esecuzione degli handler atomica
- Concettualmente:

ATTORI = OGGETTI + CONCORRENZA

- Vari framework e linguaggi ad attori disponibili
  - es: Framework Akka (akka.io), ActorFoundry (academics)

PROGRAMMAZIONE AD ATTORI



ESEMPIO IN ACTORFOUNDRY

```
public class PingActor extends Actor {
  ActorName otherPinger;
  @message
  public void start(ActorName other) {
    otherPinger = other;
    send(otherPinger, "ping", self(), Id.stamp()+"called from " + self());
  }
  @message
  public void ping(ActorName caller, String msg) {
    send(stdout, "println", Id.stamp()+"Received ping (" + msg +") from " + caller + "...");
    send(caller, "alive", Id.stamp()+self().toString() + " is alive");
  }
  @message
  public void alive(String reply) {
    send(stdout, "println", Id.stamp()+"Received " + reply + " from pinged actor");
  }
}
```

```
public class PingBoot extends Actor {

  @message
  public void boot()  throws RemoteCodeException {
    ActorName pinger1 = null;
    ActorName pinger2 = null;

    pinger1 = create(osl.examples.ping.PingActor.class);
    pinger2 = create(osl.examples.ping.PingActor.class);

    send(pinger1, "start", pinger2);
  }
}
```

BIBLIOGRAFIA PER APPROFONDIMENTI

- Concurrent Programming in Java: Design Principles and Pattern, 2/E - Doug Lea - Addison-Wesley Professional
  - testo di riferimento per la programmazione concorrente in Java
- Java Concurrency in Practice - Brian Goetz et al - Addison Wesley