Sincronizzazione e meccanismi adottati in JAVA

Advanced Computer Programming

Prof. Luigi De Simone

Sommario

- Sincronizzazione in Java
- Monitor in Java
- Wait e Notify
- Meccanismi di sincronizzazione in Java 1.5

Riferimenti:

- Bruce Eckel, “Thinking in Java” capitolo 13
- Tutorial Java su concorrenza e sincronizzazione - http://java.sun.com/docs/books/tutorial/essential/concurrency/sync.html

Sincronizzazione in Java

- Java fornisce il meccanismo di sincronizzazione basato sui **mutex** per l'accesso a sezione critiche
- Ogni **oggetto** ha associato un mutex
- Il mutex di un qualsiasi oggetto non è acceduto direttamente dall'applicazione, ma tramite l'utilizzo di:
  - **metodi sincronizzati**
  - **blocchi sincronizzati**

Sincronizzazione in Java

- Quando un thread esegue un metodo/blocco che è stato dichiarato **sincronizzato (synchronized)**:
  - entra in possesso del mutex associato all'istanza;
  - il thread che blocca il mutex ("mutex lock") acquisisce l'accesso esclusivo alla sezione critica;
  - eventuali thread che vogliano accedere alla sezione critica saranno posti in uno stato di attesa.

Sincronizzazione in Java

- **Metodo sincronizzato**
  - anteponendo la parola chiave synchronized alla firma del metodo:

```
synchronized void Method(){...}
```

- L'accesso al metodo è effettuato solo quando il **lock associato all'oggetto è stato acquisito**

Synchronized: esempio

```
class SharedCounter{

        private int theData;

        public SharedCounter(int initialValue){
                theData=initialValue;
        }

        public synchronized int read(){
                return theData;
        }
}
```

Acquisisce il **lock** sull'oggetto.

Rilascia il **lock** sull'oggetto

Note

- I metodi non sincronizzati non richiedono il lock e possono essere eseguiti in ogni istante **senza garanzie di mutua esclusione.**
- I **metodi sincronizzati** garantiscono l'accesso in mutua esclusione ai dati incapsulati in un oggetto **solo se si accede ai dati tramite metodi dichiarati synchronized**

Sincronizzazione metodi statici

- **Anche i metodi statici possono essere dichiarati sincronizzati**
  - poiché essi non sono legati ad alcuna istanza, **viene acquisito il mutex associato alla classe**
- Se invochiamo due metodi statici sincronizzati di una stessa classe da due threads diversi
  - essi verranno eseguiti **in sequenza**
- Se invochiamo un metodo statico e un metodo di istanza, entrambi sincronizzati, di una stessa classe
  - essi verranno eseguiti **in concorrenza**

Sincronizzazione metodi statici

```
Class StaticSharedVariable{

      public synchronized int Read(){
            …
      }
      public synchronized static void Write (int i){
            …
      }

…
}
```

- Ottenere il lock della classe non influenza i lock di qualsiasi istanza della classe.

Vantaggi per il programmatore

Il programmatore **non ha la preoccupazione di rilasciare il mutex** ogni volta che un metodo termina normalmente o per una eccezione, viene eseguito **automaticamente**.

Blocchi synchronized

- Java offre la possibilità di definire sezioni critiche anche quando queste non coincidono con il corpo di un metodo attraverso i cosiddetti **blocchi sincronizzati**.
- La parola chiave **`synchronized`** prende come parametro **il riferimento ad un oggetto** del quale si deve ottenere il lock per continuare

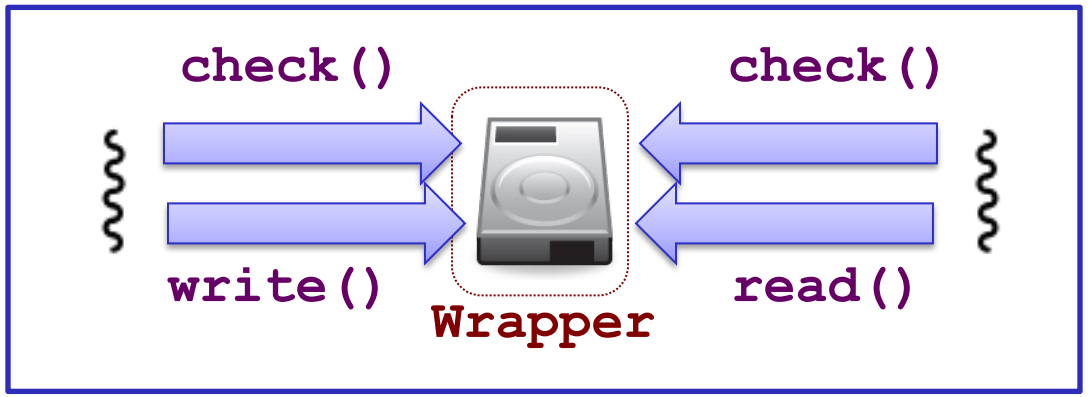
```
public int read(){
      synchronized(this) //oggetto corrente
      {
            return theData;
      }
}
```

Blocchi synchronized: avvertenze

- **Maggiore espressività nell'implementare vincoli di sincronizzazione di un programma**
- L'eccessivo uso di blocchi sincronizzati può rendere il codice disordinato
  - I vincoli di sincronizzazione non sono più incapsulati in un singolo posto (es. definizione di un metodo)
  - Non è possibile comprendere i vincoli di sincronizzazione associati ad un oggetto "O" guardando all'oggetto "O".
    - **bisogna guardare a tutti gli oggetti che accedono ad "O" in un blocco synchronized**

Es. sincronizzazione (1/4)

- Es: un insieme di thread che accede in lettura e/o scrittura ad una variabile in maniera concorrente:
  - un thread deve eseguire le seguenti operazioni in mutua esclusione:
    - **check() - Controllo disponibilità**
    - **write() / read() – Operazioni di scrittura / lettura.**
  - **Vincolo:** Se un thread ha **invocato check()**, deve avere la **certezza che nessun altro thread possa usare la risorsa** prima che possa eseguire una **read()** o una **write()**



Es. sincronizzazione (2/4)

- **Implementazione con soli metodi synchronized**

```
class Wrapper
{
        private int buffer;
        .
        .
        .
        synchronized void write() {. . .}
        synchronized void read() {. . .}
        synchronized bool check() {. . .}
}
```

operazioni di scrittura e lettura in mutua esclusione.

- Non è corretta:
  - Un thread può acquisire il monitor rilasciato da un altro all'uscita di check(), **andando a violare il requisito di atomicità della coppia check() + read()/write().**

Es. sincronizzazione (3/4)

È corretto ricorrere ai blocchi sincronizzati:

```
class Wrapper
{
        private int buffer;
        . . .
        void write() {. . .}
        void read() {. . .}
        bool check() {. . .}
}
```

```
class MyWriter extends Thread{
    private Wrapper wrapper;
    . . .
    public MyWriter(Wrapper _wrapper){
        wrapper = _wrapper;
    }
    void run() {
        synchronized(wrapper) {
                . . .
            if(wrapper.check())
                wrapper.write();
        }
    }
}
```

Implementiamo anche una classe MyRead, sottoclasse di Thread, che nel metodo run() invoca read(), invece di write().

Es. sincronizzazione (3/4)

È corretto ricorrere ai blocchi sincronizzati:

```
class Wrapper
{
        private int buffer;
        . . .
        void write() {. . .}
        void read() {. . .}
        bool check() {. . .}
}
```

Nella classe wrapper non adottiamo nessuna particolare misura di sincronizzazione.

```
class MyWriter extends Thread{
    private Wrapper wrapper;
    . . .
    public MyWriter(Wrapper _wrapper){
        wrapper = _wrapper;
    }
    void run() {
        synchronized(wrapper) {
                . . .
            if(wrapper.check())
                wrapper.write();
        }
    }
}
```

Es. sincronizzazione (3/4)

È corretto ricorrere ai blocchi sincronizzati:

```
class Wrapper
{
        private int buffer;
        . . .
        void write() {. . .}
        void read() {. . .}
        bool check() {. . .}
}
```

```
class MyWriter extends Thread{
    private Wrapper wrapper;
    . . .
    public MyWriter(Wrapper _wrapper){
        wrapper = _wrapper;
    }
    void run() {
        synchronized(wrapper) {
                . . .
            if(wrapper.check())
                wrapper.write();
        }
    }
}
```

Nel metodo run() di ogni thread realizziamo un blocco sincronizzato usando il monitor associato all'istanza di Wrapper.

Es. sincronizzazione (4/4)

```
class TestApp
{
   public static void main() {
       . . .
       Wrapper buf = new Wrapper();
       boolean is_reader = false
       for(int i = 0; i < N; i++) {
          Thread th;
          if(is_reader) {
                th = new MyReader(buf);
                is_reader = false;
          } else {
                th = new MyWriter(buf);
                is_reader = true;
          }
          th.start();
       }
}
```

Istanzio un oggetto della classe Wrapper.

Es. sincronizzazione (4/4)

```
class TestApp
{
   public static void main() {
       . . .
       Wrapper buf = new Wrapper();
       boolean is_reader = false
       for(int i = 0; i < N; i++) {
          Thread th;
          if(is_reader) {
                th = new MyReader(buf);
                is_reader = false;
          } else {
                th = new MyWriter(buf);
                is_reader = true;
          }
          th.start();
       }
}
```

Istanzio degli oggetti delle classi MyReader e MyWriter passando sempre lo stesso oggetto di Wrapper, così che tutti i thread abbiano i relativi blocchi sincronizzati sullo stesso monitor.

Sincronizzazione implicita

- Se una classe non ha metodi sincronizzati ma si desidera evitare l'accesso contemporaneo a uno o più metodi
  - è possibile acquisire il mutex di una determinata istanza racchiudendo le invocazioni dei metodi da sincronizzare in un **blocco sincronizzato**

- **Struttura dei blocchi sincronizzati**

```
private Object obj=new Object();
synchronized ( obj ) {
        comando 1;
        ...
        comando n;
}
```

Monitor in JAVA

- Come spiegato, **synchronized** indica che il thread deve acquisire il lock sull'oggetto
- Una classe con metodi **synchronized** è un **monitor in Java**
- **Un monitor Java ha una sola (ed implicita) variabile condition**
  - Il thread attivo nel monitor può sospendere la propria esecuzione invocando la primitiva **wait()**;
  - Quest'ultima ha l'effetto di rilasciare il monitor ed inserire il thread nel *wait set*
  - Il thread rimarrà nel *wait set* finché non verrà invocata una **notify()** da un altro thread che è attivo nel monitor;

Esempio (gestione del deposito)

```
public synchronized void deposita (){

        while ( disponibilita == 0 ){
                try{
                        System.out.println ( "Deposita: nessuno spazio!" );
                        wait ();
                }catch ( InterruptedException e ){
                        e.printStackTrace();
                }
        }

        disponibilita=disponibilita-1;
        System.out.println ( "Deposita disp= " + disponibilita );

        notifyAll();

}
```

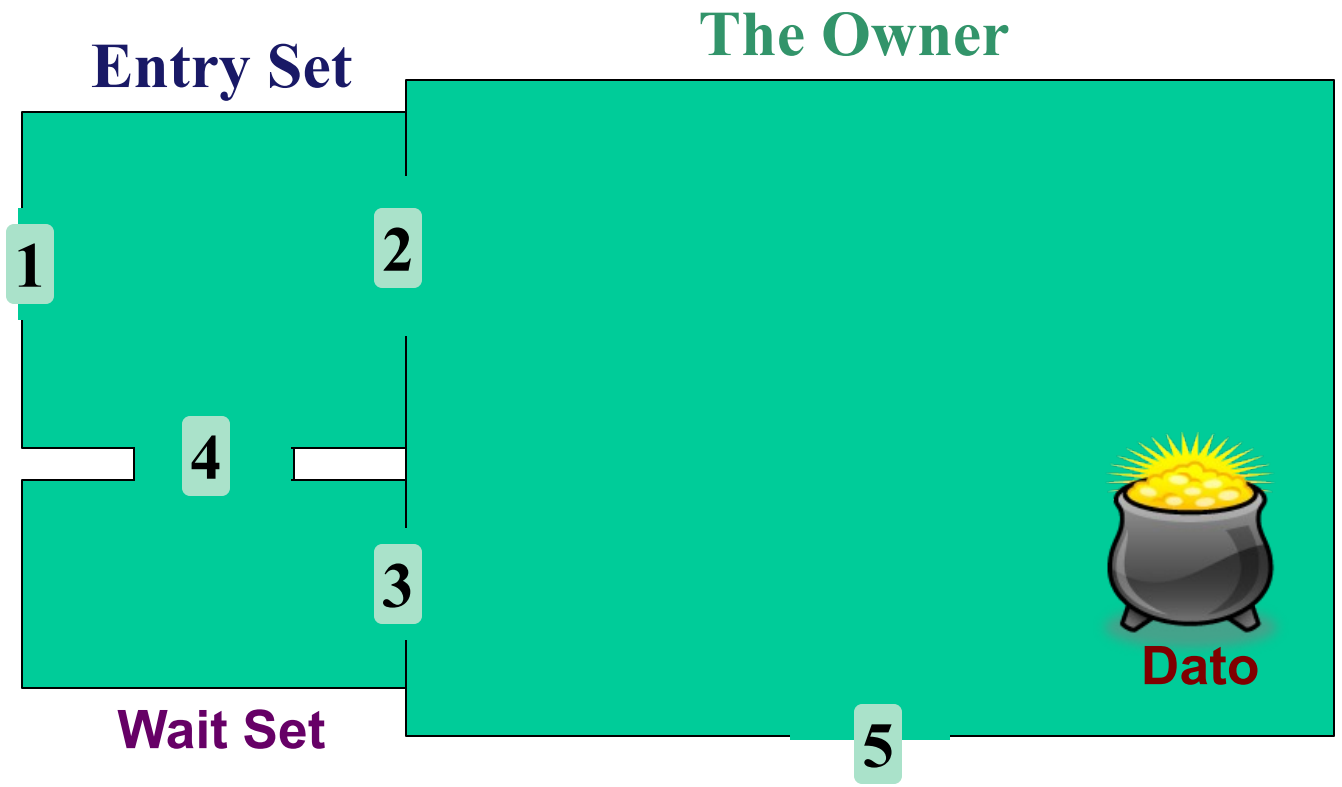
Esempio (gestione del deposito)

```
public synchronized void preleva () {

        while ( disponibilita == dimensione ){
                try{
                        System.out.println ( "Preleva: nessun oggetto!" );
                        wait();
                }catch( InterruptedException e){
                        e.printStackTrace();
                }
        }

        disponibilita=disponibilita+1;
        System.out.println ( "Preleva disp= " + disponibilita );

        notifyAll();

}
```

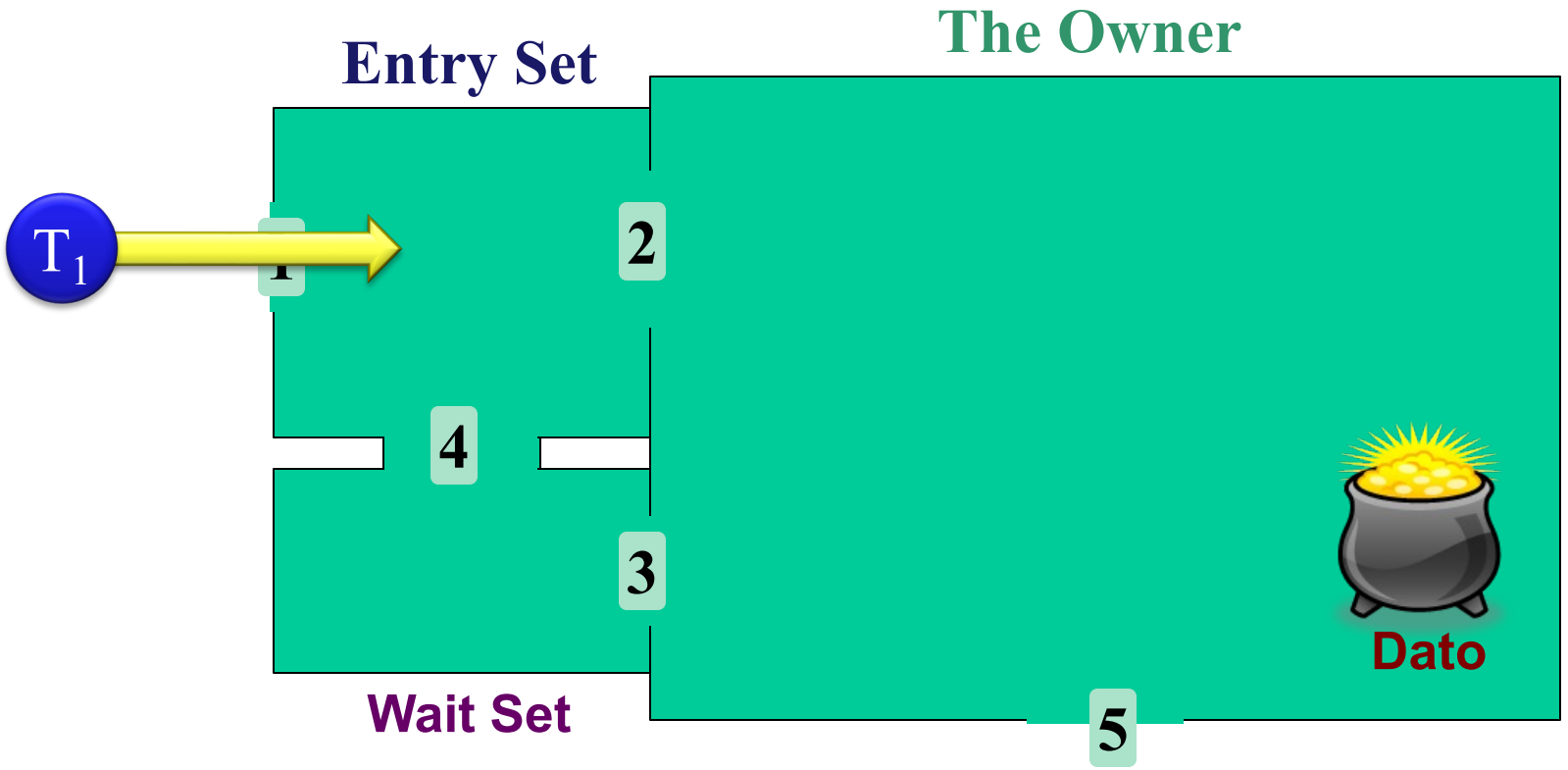
Monitor della JVM

- La soluzione adottata in Java è quella "***signal and continue***" monitor poiché
- Differentemente dalla soluzione di Hoare, un thread Java che invoca la notify() **rimane in possesso del monitor e continua la propria esecuzione** all'interno della monitor region.

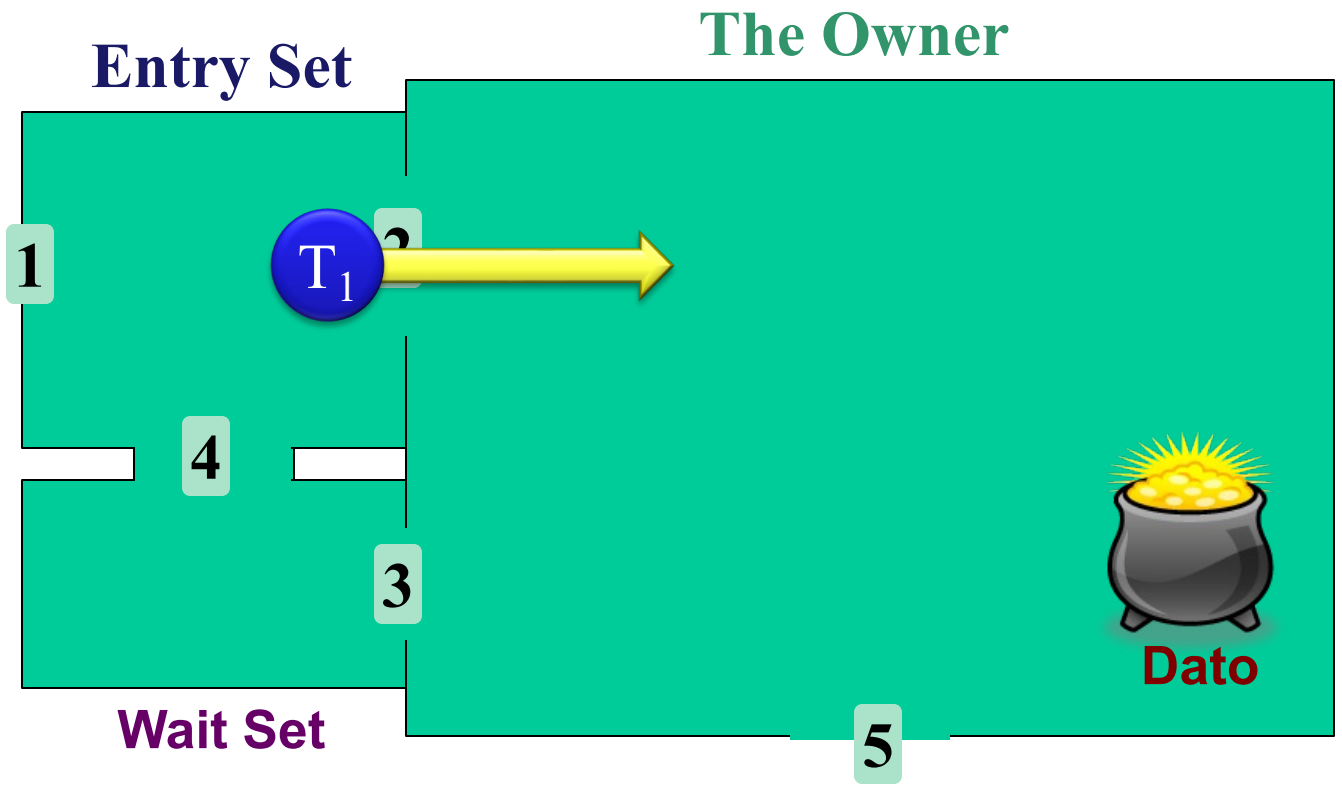
Modello di sincronizzazione



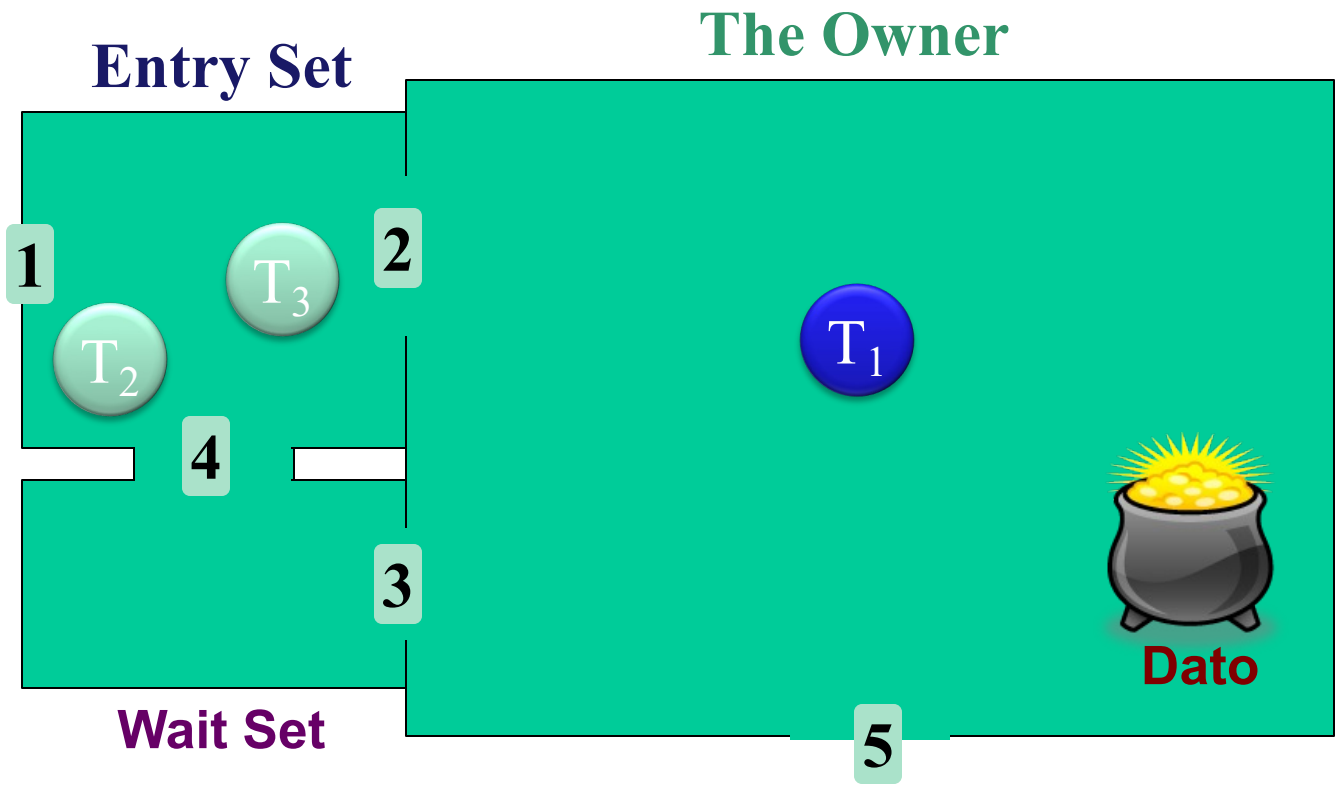
Modello di sincronizzazione



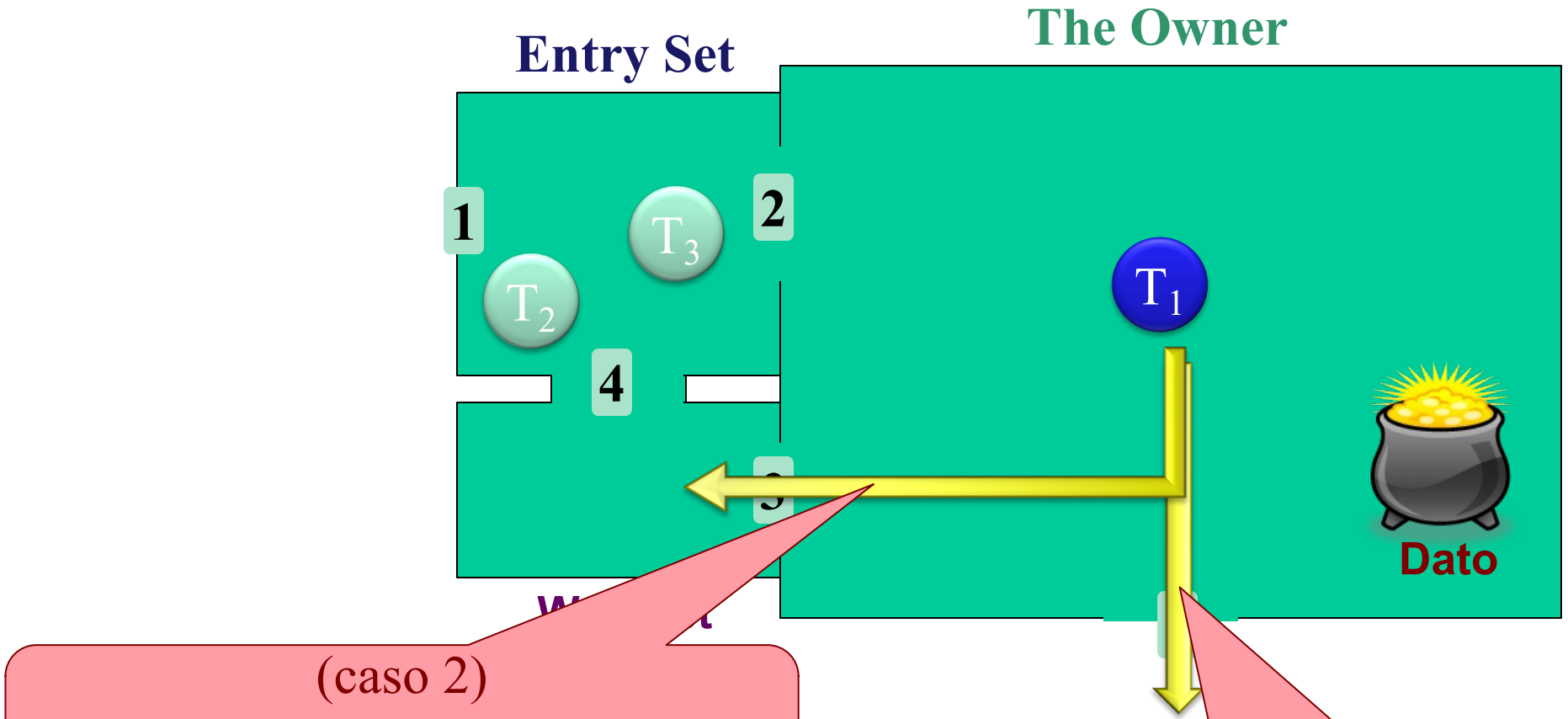
Modello di sincronizzazione



Modello di sincronizzazione



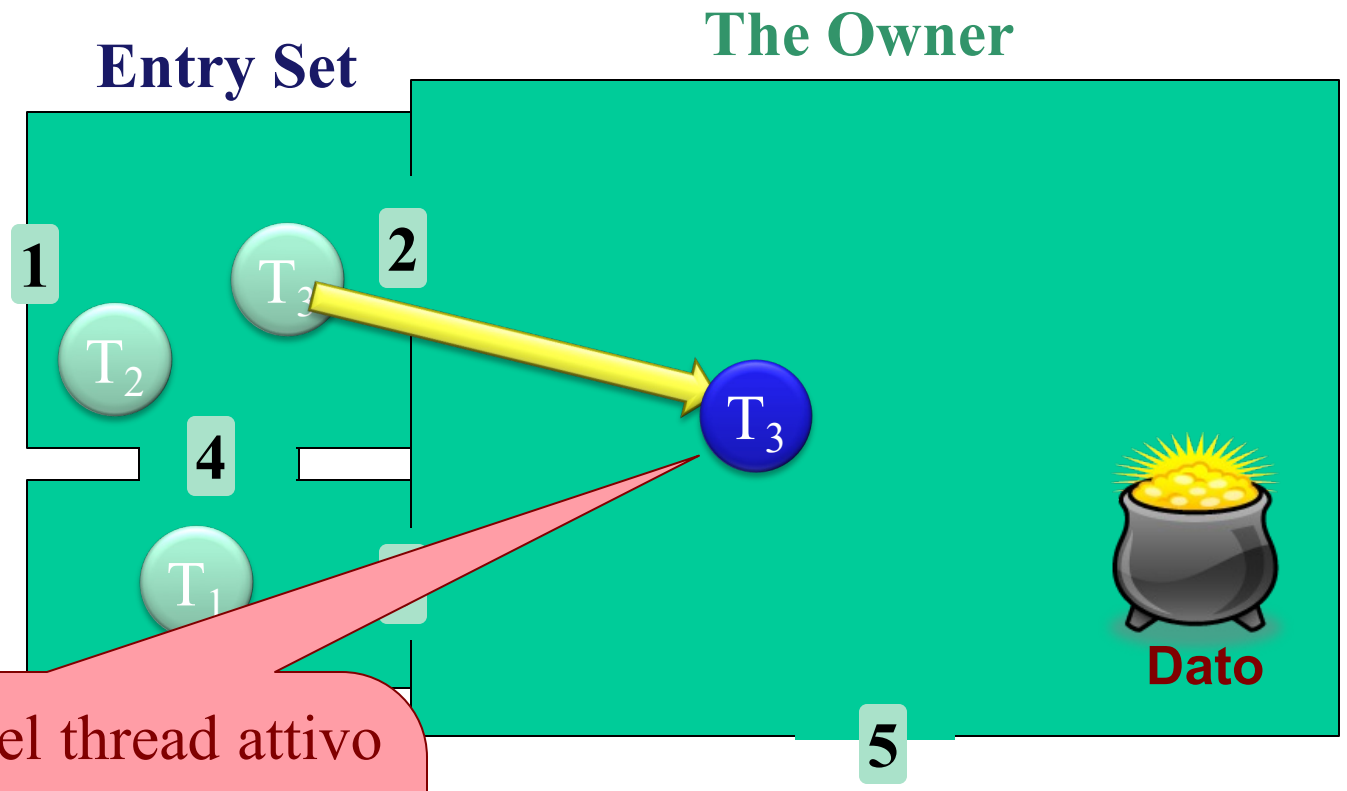
Modello di sincronizzazione



(caso 2)
Il thread attivo esegue un comando di wait().

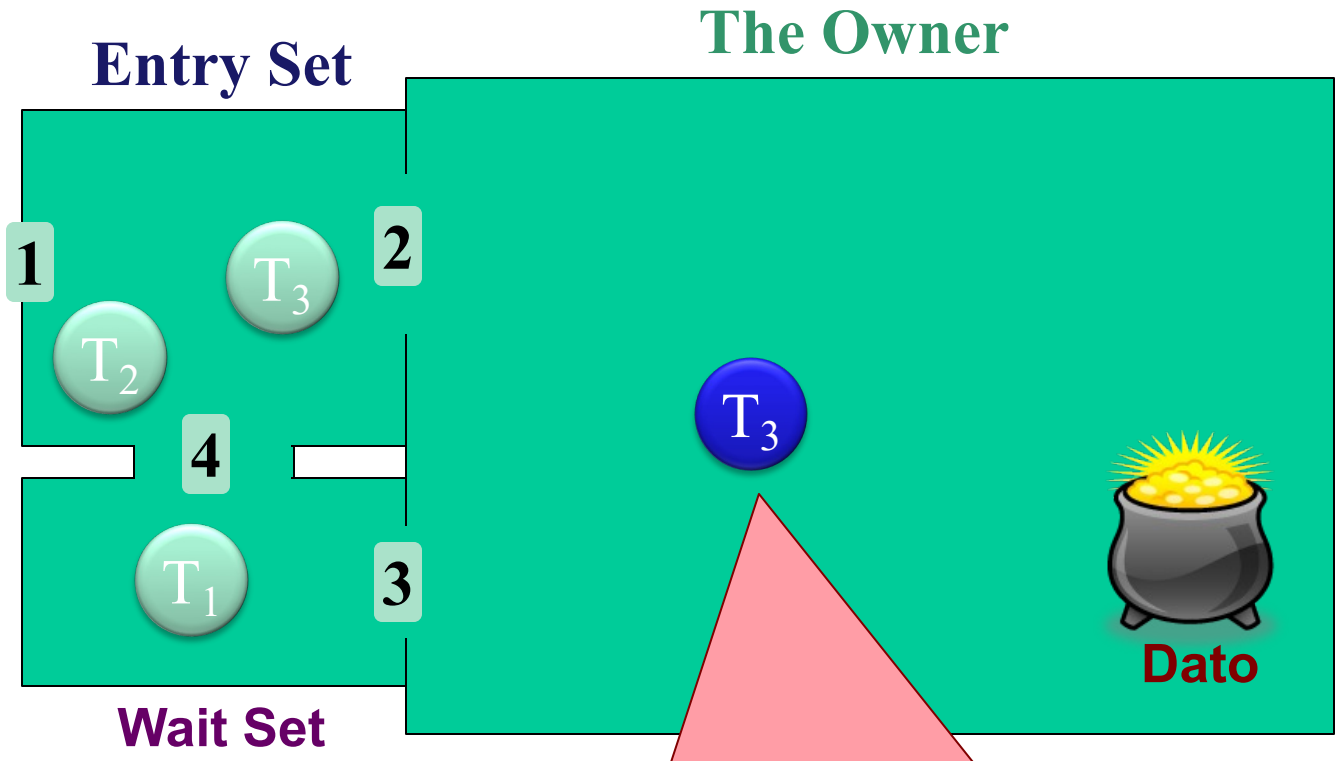
(caso 1)
Il thread attivo completa l'esecuzione delle istruzioni in sezione critica.

Modello di sincronizzazione



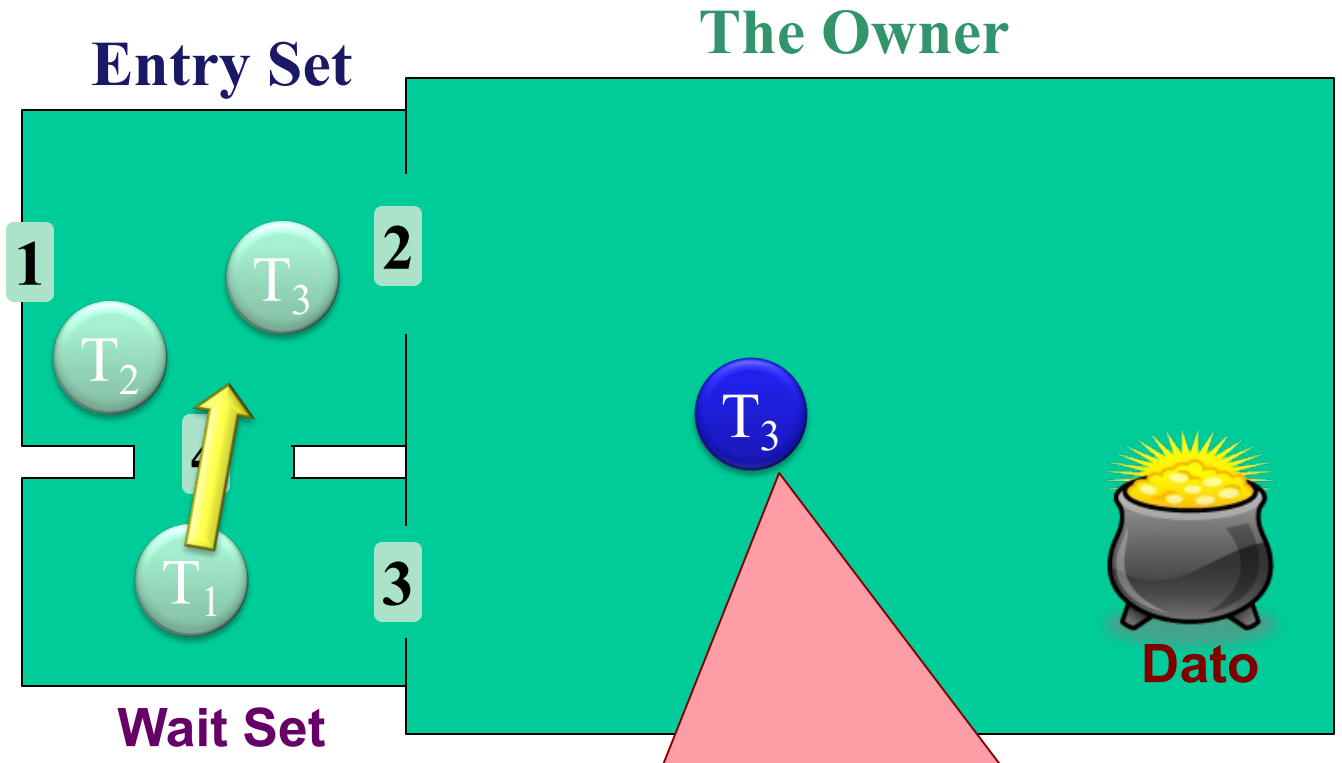
… il passaggio del thread attivo nel Wait Set comporta il rilascio del monitor, cosicché uno dei thread bloccati nell'Entry Set viene sbloccato e si avvicenda nell'uso della stanza.

Modello di sincronizzazione



Se il thread attivo **non** effettua una notify() prima di terminare l'esecuzione nella sezione critica, allora solo i thread nell'Entry Set possono competere per ottenere la risorsa.

Modello di sincronizzazione



Se il thread attivo invoca il comando notify (o notifyAll), allora un (o più di uno) thread da Wait Set si sposta in Entry Set…

Alcune considerazioni

- Al risveglio, un thread non può assumere che la sua condizione sia vera dato che altri threads potrebbero essere risvegliati indipendentemente dalla condizione sulla quale erano in attesa
- In definitiva, Java non offre garanzie che un thread risvegliato dopo una wait possa immediatamente acquisire il lock.
  - è essenziale che un thread verifichi nuovamente la sua condizione, appena risvegliato!

Cooperazione

```
…
//Sezione Critica
synchronized(obj) {
   while(<not condition>) {
       obj.wait();
   }

   do_something();
}
…
```

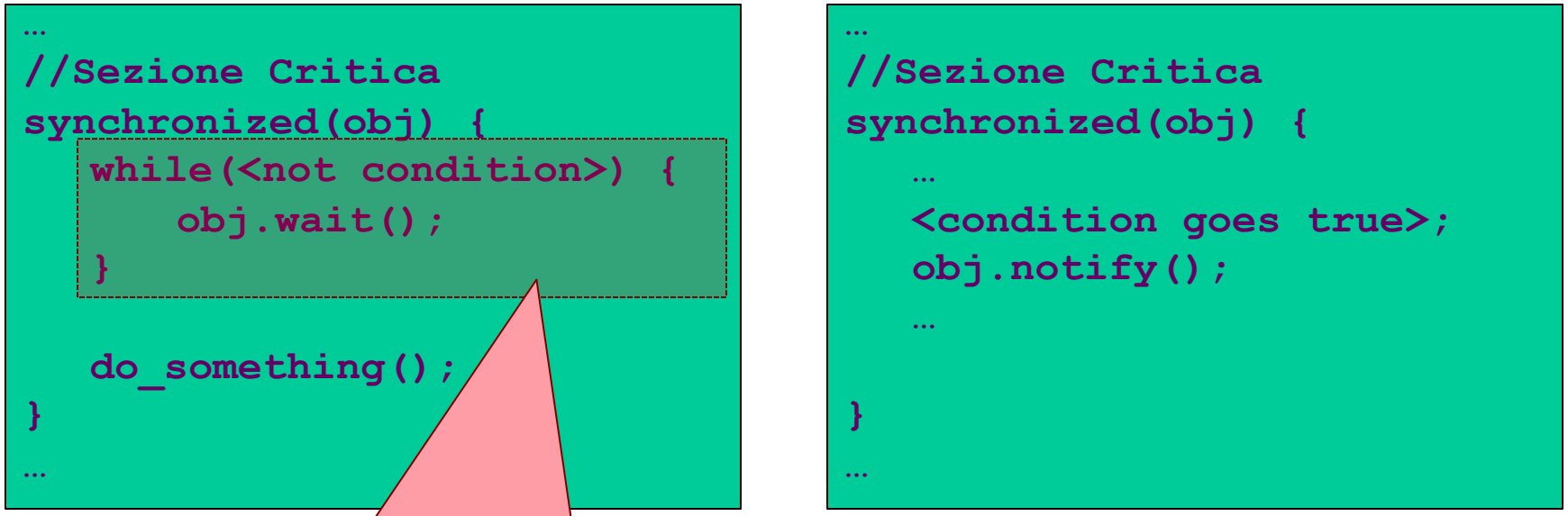
Thread 1

```
…
//Sezione Critica
synchronized(obj) {
   …
   <condition goes true>;
   obj.notify();
   …

}
…
```

Thread 3

Le primitive di cooperazione vengono ereditate dalla classe Object e possono essere invocate solo all'interno di sezioni critiche

Cooperazione



Le primitive di cooperazione vengono ereditate dalla classe Object e possono essere invocate solo all'interno di sezioni critiche

Cooperazione

```
…
//Sezione Critica
synchronized(obj) {
   while(<not condition>) {
      obj.wait();
   }

   do_something();
}
…
```

Thread 1

```
…
//Sezione Critica
synchronized(obj) {
   …
   <condition goes true>;
   obj.notify();
   …

}
…
```

Thread 3

Quando un thread abilita la condizione, sblocca anche uno dei thread che era stato posto in attesa per l'esecuzione nel metodo wait().

Cooperazione

```
…
//Sezione Critica
synchronized(obj) {
   while(<not condition>) {
       obj.wait();
   }

   do_something();
}
…
```

Thread 1

```
…
//Sezione Critica
synchronized(obj) {
   …
   <condition goes true>;
   obj.notify();
   …

}
…
```

Thread 3

I metodi wait() e notify() sono primitive di cooperazione. Non risolvono la mutua esclusione!

Wait()

- **`public final void wait( )`**
  - il thread che invoca questo metodo entra nel wait set del monitor, rilasciando il mutex associato all'istanza e rimane sospeso fintanto che non viene risvegliato da un altro thread che invoca il metodo **notify** o **notifyAll**, oppure viene interrotto con il metodo **interrupt** della classe **Thread**
- **`public final void wait (long millis)`**
  - si comporta analogamente al precedente, ma se dopo un'attesa corrispondente al numero di millisecondi specificato in **millis** non è stato risvegliato, esso si risveglia
- **`public final void wait (long millis, int nanos)`**
  - si comporta analogamente al precedente, ma permette di specificare l'attesa con una risoluzione temporale a livello di nanosecondi

Notify()

- **`public final void notify ( )`**
  - risveglia un thread nel wait set del monitor
  - poiché il metodo che invoca **notify** deve aver acquisito il mutex, il thread risvegliato deve attenderne il rilascio e competere per la sua acquisizione come un qualsiasi altro thread
- **`public final void notifyAll ( )`**
  - risveglia tutti i threads nel wait set del monitor
  - i threads risvegliati competono per l'acquisizione del mutex e se ne esiste uno con priorità più alta, esso viene subito eseguito

Meccanismi di sincronizzazione in JAVA 1.5

Meccanismi di sincronizzazione in Java 1.5

- Semafori
- Lock
- …

Riferimenti:

- Java API Specification
- Overview delle utility di sincronizzazione – http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/overview.html

Introduzione

- I monitor built-in Java sono **semplici** da utilizzare, ma hanno diverse limitazioni, tra cui:
  - **unica wait-queue;**
  - non esiste la possibilità di ***tentare*** semplicemente **di acquisire un lock**
- Tecnicamente è possibile realizzare semafori e/o monitor a partire dai costrutti primitivi Java

Java 1.5 Concurrency Utilities

- A partire da Java 1.5, sono stati introdotti ulteriori costrutti per la gestione della sincronizzazione
- Tre packages:
  - `java.util.concurrent` - fornisce le classi per supportare paradigmi comuni di programmazione concorrente
  - `java.util.concurrent.atomic` - fornisce un supporto per una programmazione lock-free e thread-safe su variabili semplici come atomic integer, atomic boolean, ecc.
  - `java.util.concurrent.locks` - fornisce un framework per diversi algoritmi di locking
    - Es., read-write locks e condition variables.

Vantaggi

- **Riuso**: minore sforzo di programmazione
- Maggiori **prestazioni** e **affidabilità**
- Package **standard** e testati
- **Debugging** più semplice

Semafori

- La classe Semaphore gestisce un insieme di permessi.
  - Ogni **acquire()** eseguito sul semaforo blocca il thread corrente se sul semaforo non c'è almeno un permesso disponibile da acquisire.
  - Viceversa, una **release()** aggiunge un permesso al semaforo, potenzialmente sbloccando un thread bloccato in fase di acquisizione.
  - **boolean tryAcquire()** acquisisce un permesso sul semaforo, solo se il permesso è disponibile al momento dell'invocazione.

- I semafori vengono utilizzati generalmente per *limitare* il numero di thread che accedono ad una determinata risorsa condivisa.
  - NB: sebbene il semaforo incapsula la sincronizzazione necessaria a restringere l'accesso alla risorsa, la consistenza della risorsa va comunque gestita a parte.

Semafori

- Un semaforo inizializzato ad “1” ed usato in modo da non avere più di “1” permesso disponibile, funge da **lock per la mutua esclusione** (semaforo binario).

- A differenza di molte implementazioni di Lock (presenti nel package `java.util.concurrent.locks`), il lock effettuato col semaforo binario **può** essere rilasciato da un qualsiasi altro thread.

- **Il costruttore della classe Semaphore ha come parametro il numero di permessi iniziale**, inoltre ha un parametro opzionale boolean per imporre l’ordinamento FIFO (`true`) o non garantire alcun ordinamento (`false` o non specificato):
  - `Semaphore(int permits)`
  - `Semaphore(int permits, boolean fair)`

Esempio

```
class Pool {
    private static final MAX_AVAILABLE = 100;
    private Semaphore available = new Semaphore(MAX_AVAIL, true);

    public Object getItem() throws InterruptedException {
          available.acquire();
          return getNextAvailableItem();
    }

    public void putItem(Object x) {
           if (markAsUnused(x)) available.release();
    }

    protected Object[] items = ...
    protected boolean[] used = new boolean[MAX_AVAILABLE];

    protected synchronized Object getNextAvailableItem()
      { . . .}

    protected synchronized boolean markAsUnused(Object item)
      { . . . }
}
```

Esempio

```
class Pool {
    private static final MAX_AVAILABLE = 100;
    private Semaphore available = new Semaphore(MAX_AVAIL, true);

    }

    public void putItem(Object x) {
            if (markAsUnused(x)) available.release();
    }

    protected Object[] items = ...
    protected boolean[] used = new boolean[MAX_AVAILABLE];

    protected synchronized Object getNextAvailableItem()
      { . . .}

   protected synchronized boolean markAsUnused(Object item)
     { . . . }
}
```

Istanzio un oggetto semaforo con numero di permessi pari a MAX_AVAIL e con politica di accodamento FIFO.

Esempio

```
class Pool {
    private static final MAX_AVAILABLE = 100;
    private Semaphore available = new Semaphore(MAX_AVAIL, true);

    public Object getItem() throws InterruptedException {
        available.acquire();
        return getNextAvailableItem();
    }

    protected Object[] items = ...
    protected boolean[] used = new boolean[MAX_AVAILABLE];

    protected synchronized Object getNextAvailableItem()
      { . . .}

    protected synchronized boolean markAsUnused(Object item)
      { . . . }
}
```

Acquisisco un permesso sul semaforo, tale metodo può sollevare eccezione di tipo InterruptedException (come già visto in merito a wait()).

Esempio

```
class Pool {
    private static final MAX_AVAILABLE = 100;
    private Semaphore available = new Semaphore(MAX_AVAIL, true);

    public Object getItem() throws InterruptedException {
          available.acquire();
          return getNextAvailableItem();
    }

    public void putItem(Object x) {
           if (markAsUnused(x)) available.release();
    }

    protected synchronized Object getNextAvailableItem()
      { . . .}

    protected synchronized boolean markAsUnused(Object item)
      { . . . }
}
```

Rilascio un permesso sul semaforo, tale metodo non solleva alcun tipo di eccezione (come già visto in merito a notify()).

Lock

- Lock è un'interfaccia del package **java.util.concurrent**

- Le sue implementazioni offrono un uso più flessibile rispetto ai lock associati ai blocchi o metodi synchronized.

- Permettono l'accesso esclusivo ad una risorsa

```
Lock l = ...;
  l.lock();
  try {
      // accesso alla risorsa protetta da questo lock
  } finally {
      l.unlock();   //garantire il rilascio della risorsa
  }
```

Lock

- Le implementazioni di Lock offrono funzionalità addizionali oltre a quelli fornite dall'uso dello statement synchronized:
  - Un tentativo non bloccante di acquisire il lock (tryLock());
  - Un tentativo di acquisire il lock che può essere interrotto (lockInterruptibly()). Un altro thread può interrompere il thread in attesa del lock invocando il metodo interrupt();
  - Un tentativo di acquisire il lock che può essere interrotto da timeout (tryLock(long, TimeUnit)).
- La classe ReentrantLock è un Lock con lo stesso comportamento e semantica dei lock associati ai metodi e blocchi synchronized.

Lock

- I Lock vengono adoperati per la gestione dell'accesso in mutua esclusione, ma offrono meccanismi anche per la cooperazione tra thread attraverso implementazioni dell'interfaccia **Condition**.
- Condition fornisce un mezzo per un thread di sospendere l'esecuzione fino a quando è notificato da un altro thread che una certa condizione di stato è ora vera.
- Gli oggetti di tipo Condition consentono di implementare le **condition variables.**
- Un'istanza di tipo Condition è intrinsecamente legata a un Lock. Per ottenere un'istanza Condition per una particolare istanza di Lock bisogna usare il suo metodo **newCondition()**.

Lock

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
   }
. . .
```

Lock

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
   }
. . .
```

Istanzio un oggetto della classe ReentrantLock.

Lock

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
   }
. . .
```

Ottengo due oggetti Conditions dal lock.

Lock

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
   }
. . .
```

Obbligo il chiamante a gestire le eccezioni sollevabili dalla wait.

Lock

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
                                          utptr = 0;
     }
   }
. . .
```

Proteggo l'accesso in sezione critica.
E mi metto in attesa che la condizione sia vera.

Lock

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
```

Abilito la seconda condizione e sblocco il lock.

Lock

```
. . .

   public Object take() throws InterruptedException {
     lock.lock();
     try {
       while (count == 0)
         notEmpty.await();
       Object x = items[takeptr];
       if (++takeptr == items.length) takeptr = 0;
       --count;
       notFull.signal();
       return x;
     } finally {
       lock.unlock();
     }
   }
 }
```

Lock

```
. . .
  public Object take() throws InterruptedException {
    lock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      Object x = items[takeptr];
      if (++takeptr == items.length) takeptr = 0;
      --count;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
```

Metodo take() che acquisisce il lock, controlla la condizione e poi sblocca il lock.