

# Programmazione concorrente e threads

---

## Programmazione concorrente

---

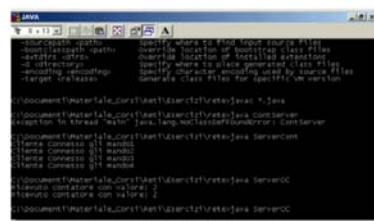
- Un programma concorrente è un programma nel quale diverse computazioni possono essere eseguite allo stesso tempo
- Alcuni sistemi sono implicitamente concorrenti:
  - Concorrenza in database query
  - caching delle istruzioni e branch prediction nelle CPUs
  - scheduling nei sistemi operativi
- Utilizzo esplicito di concorrenza all'interno dei programmi:
  - Programmazione concorrente → scrittura di programmi che utilizzano istruzioni esplicitamente parallele
  - Java fornisce il supporto per la concorrenza
  - C non ha un supporto per la concorrenza ma usa chiamate di sistema operativo.
  - Altri linguaggi meno popolari forniscono un supporto esplicito per la concorrenza

## Processo

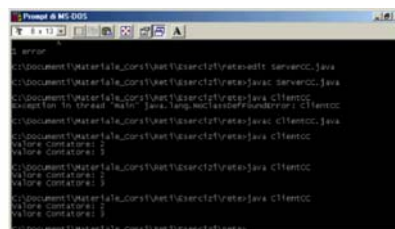
- ❑ Nei sistemi concorrenti (multiprogrammati), il termine **processo** denota la *singola attività* eseguita in modo indipendente e separato rispetto alle altre.
- ❑ Tipicamente, un programma è eseguito da un processo.
  - Il processo è l'istanza in esecuzione di un processo
  - Più processi possono eseguire lo stesso programma.
- ❑ **Un processo:**
  - è del tutto autonomo e indipendente dagli altri processi
  - ha un proprio *spazio di indirizzamento riservato* (porzione di memoria RAM) e non condivide memoria con altri processi: significa che anche le variabili globali sono private al processo
  - ha un proprio *stack riservato* (quindi anche le variabili locali sono private al processo).
- ❑ La comunicazione tra processi distinti non può quindi avvenire in modo "diretto" (condivisione di variabili), ma deve sfruttare appositi meccanismi di sistema (p.e., file o socket).
- ❑ I processi servono ad avere attività in esecuzione concorrente ma indipendente, minimizzando le possibilità di interazione, e quindi di disturbo reciproco

## ESEMPIO

- ❑ Quando da una finestra lancio un server, e poi da un'altra finestra lancio un client, questi due programmi in esecuzione sono processi diversi, e non possono interagire se non tramite la scrittura di parti comuni di file o attraverso comunicazione via socket!!!



```
javac -d bin src\com\example\server\Server.java
java -cp bin com.example.server.Server
[main] INFO com.example.server.Server - Server started
[main] INFO com.example.server.Server - Listening for connections on port 8080
```



```
javac -d bin src\com\example\client\Client.java
java -cp bin com.example.client.Client
[main] INFO com.example.client.Client - Client started
[main] INFO com.example.client.Client - Connecting to server on port 8080
[main] INFO com.example.client.Client - Connected
[main] INFO com.example.client.Client - Sending message: Hello from client
```

## THREAD

---

- ❑ Spesso si vogliono realizzare programmi in cui non vi è solo una attività in esecuzione, ma in cui diverse attività in concorrenza eseguono (motivi di efficienza dell'esecuzione). In questo caso, vi è può l'esigenza forte di fare interagire le diverse attività in esecuzione.
- ❑ **Un thread:**
  - è un'attività autonoma che **"vive" all'interno di un processo** (e quindi, tipicamente, la stessa istanza in esecuzione di un programma, corrispondente a un processo, ospita al suo interno dei sotto-flussi di esecuzione concorrente, corrispondenti ai thread)
  - **...ma non ha uno spazio di indirizzamento riservato: tutti i thread appartenenti allo stesso processo condividono il medesimo spazio di indirizzamento.** (le variabili globali in RAM)

## THREAD

---

- ❑ **Un thread:**
  - ha un proprio *stack riservato* (e quindi le sue variabili locali di una procedura sono private e non condivise da altri thread) La comunicazione fra thread può quindi avvenire in modo molto più semplice, tramite lo spazio di memoria condiviso. Quindi, se ci sono variabili globali o riferimenti a aree di memoria od oggetti comuni diversi thread possono interazione.
- ❑ Occorre però *disciplinare l'accesso* a tale area comune
  - necessità di opportuni *meccanismi di sincronizzazione*.
- ❑ In un linguaggio ad oggetti: ovviamente, il concetto di variabile globale non ha senso. Però più thread possono condividere riferimenti allo stesso oggetto, interagiscono tramite tale oggetto.

## Che cosa è un Thread?

---

- Un programma sequenziale ha
  - un inizio
  - una sequenza di esecuzione
  - Una fine.
  - Ad ogni istante durante la sua esecuzione c'è un singolo punto in esecuzione

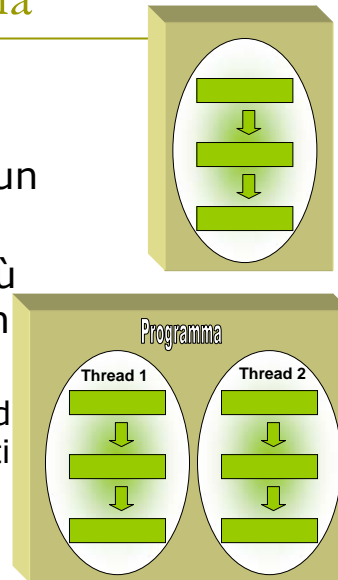
## Thread vs. Programmi sequenziali

---

- Un thread è simile a un programma sequenziale
- Un singolo thread ha
  - una sequenza di esecuzione
  - Una fine.
  - Ad ogni istante durante l'esecuzione del thread c'è un singolo punto in esecuzione
- Comunque, non è esso stesso un programma;
  - Non può essere eseguito in modo indipendente.
  - Viene eseguito all'interno di un programma.

## Threads nei programma

- Un thread è un singolo flusso sequenziale di istruzioni all'interno di un programma.
- Si possono utilizzare più threads all'interno di un singolo programma
  - Run allo stesso tempo ed eseguono tasks differenti



## Threads

- Alcuni si riferiscono ai threads come "processo leggero" perchè:
  - viene eseguito all'interno di un contesto di un programma completo
  - utilizza le risorse allocate per il programma e l'ambiente del programma
- Come un flusso di controllo sequenziale, un thread deve ritagliarsi alcune delle sue risorse all'interno del programma in esecuzione.
  - Esso deve avere per esempio il suo proprio stack di esecuzione e un program counter.
- Il codice eseguito all'interno del thread opera solo all'interno di tale contesto
- Alcuni usano il termine "*execution context*" come sinonimo per thread.

## Threads

---

- ▣ attraverso i threads è possibile in Java eseguire diversi tasks in modo concorrente (multithreading)
- ▣ un thread è essenzialmente un flusso di controllo
- ▣ threads diversi all'interno della stessa applicazione (programma) condividono la maggior parte dello stato
  - sono condivisi l'ambiente delle classi e la heap
  - ogni thread ha un proprio stack delle attivazioni
  - per quanto riguarda le variabili
    - ▣ sono condivise le variabili statiche (classi) e le variabili di istanza (heap)
    - ▣ non sono condivise le variabili locali dei metodi (stack)

## Switch di contesto

---

- ▣ in generale, quando diversi processi (flussi di esecuzione) condividono un unico processore, il sistema operativo deve ogni tanto sospendere l'esecuzione di un processo e riattivarne un altro
- ▣ si realizza con una sequenza di eventi chiamata switch di contesto
  - bisogna salvare una notevole quantità di informazione relativa al processo sospeso e ripristinare una simile quantità di informazione per il processo da riattivare
  - uno switch di contesto tra due processi può richiedere l'esecuzione di migliaia di istruzioni

## Threads e switch di contesto

---

- ▣ lo switch di contesto tra threads di un programma Java viene effettuato dalla JVM (Java Virtual Machine)
  - i threads condividono una gran parte dello stato
  - lo switch di contesto fra due threads richiede tipicamente l'esecuzione di meno di 100 istruzioni

## Thread di esecuzione

---

- ▣ Una sequenza di istruzioni

```
for (int j = 0; j < 2; j++) { // line 1
    System.out.println("hello!"); // line 2
}
```
- ▣ La sequenza di esecuzione 1,2,1,2
- ▣ Così, fino ad ora abbiamo considerato solo programmi **single-threaded**
  - La JVM esegue il programma eseguendo una singola sequenza di istruzioni.
  - Nelle applicazioni Java, il thread inizia l'esecuzione con la prima istruzione del `main`.

## Perchè evitare I threads

---

- ▣ Difficili da usare
- ▣ Rendono I programmi difficili da debuggare
- ▣ Deadlock
  - Bisogna avere cura che tutti I threads creati non invochino componenti di Swing
    - ▣ I componenti di Swing non sono "thread safe"

## I Threads sono inevitabili?

---

- ▣ Miglioramento delle performance dei programmi
  - applicazioni grafiche
- ▣ Semplificazione del codice e dell'architettura del programma
  - Per eventi temporizzati piuttosto che eseguirli dei cicli e il polling del sistema è possibile utilizzare una classe Timer che notifica il fatto che una certa quantità di tempo è trascorsa



## Uso di threads

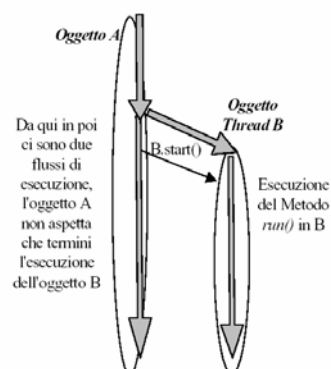
- Alcuni tipici usi dei threads sono:
  - Spostare task di inizializzazione time-consuming all'esterno del thread, così che la GUI diventi più veloce.
  - Spostare un time-consuming task all'esterno del event-dispatching thread, così che la GUI rimane interattiva
  - Per eseguire operazioni ripetitive usualmente con un predeterminato periodo di tempo fra le operazioni.
  - Aspettare per messaggi da altri programmi

## THREAD IN JAVA

Come si può realizzare il concetto di Thread in Java?

- **NEL MODO PIU' NATURALE!** Sono oggetti particolari ai quali si richiede un servizio (chiamato `start()`) corrispondente al lancio di una attività, di un thread!  
**MA:** non si aspetta che il servizio termini, esso procede in concorrenza a chi lo ha richiesto!

*Richiesta di Servizio `start()` a un Thread*



## Creare un Thread

---

- ❑ Un programma diventa multithreaded se il singolo thread costruisce e starts un secondo thread di esecuzione.
- ❑ In Java I threads sono oggetti reali.
- ❑ 2 modi per creare un thread:
  - Implementare l'interfaccia Runnable (e passare questo nel costruttore di un Thread),
  - Estendere `java.lang.Thread` (che implementa Runnable)
- ❑ Ogni thread può attivare (start) un qualsiasi numero di Threads.

## La classe Thread

---

- ❑ la classe `Thread` nel package `java.lang` ha le operazioni per creare e controllare threads in programmi Java
  - l'esecuzione di codice Java avviene sempre sotto il controllo di un oggetto `Thread`
- ❑ un oggetto di tipo `Thread` deve essere per prima cosa associato al metodo che vogliamo lui esegua
  - Java fornisce due modi per associare un metodo ad un `Thread` (vedi dopo)

## Specifica della classe Thread

---

```
public class java.lang.Thread extends
    java.lang.Object implements java.lang.Runnable
{

    /*un Thread è un oggetto che ha il controllo
    dell'esecuzione di un thread costruttori */

}
```

## Costruttori della classe thread

---

- public **Thread()**
  - Ha lo stesso effetto di Thread(null, null, *gname*), dove ***gname*** è un nuovo nome generato automaticamente. I nomi generati automaticamente sono nella forma "Thread-" + *n*, dove *n* è un intero.
- public **Thread(Runnable target)**
  - Ha lo stesso effetto di Thread(null, target, *gname*), dove ***gname*** è un nuovo nome generato automaticamente.
  - **Parameters:**
    - target – l'oggetto il cui metodo run è chiamato.

---

public **Thread**(**String** name)

- Alloca un nuovo oggetto Thread. Questo costruttore ha lo stesso effetto di Thread(null, null, name).

■ **Parameters:**

- name – il nome del nuovo thread.

public **Thread**(**Runnable** target, **String** name)

- Alloca un nuovo oggetto Thread. Questo costruttore ha lo stesso effetto di Thread(null, target, name).

■ **Parameters:**

- name – il nome del nuovo thread
- target – l'oggetto il cui metodo run è chiamato.

## Specifica (parziale) della classe Thread

---

public void start ()

*/\* fa in modo il thread possa essere schedulato per l'esecuzione; il codice da eseguire è il metodo run() dell'oggetto Runnable specificato durante la creazione; se questo non esiste è il metodo run() del thread può essere eseguito una sola volta \*/*

public void run ()

*/\* non fa niente; deve essere ridefinito in una sottoclasse di Thread oppure in una classe che implementa Runnable \*/*

## Specifica (parziale) della classe Thread

---

**public static void sleep(long n) throws  
InterruptedException**

*/\* fa dormire il thread attualmente in esecuzione (che  
mantiene i suoi locks), e non ritorna prima che siano  
trascorsi n millisecondi; solleva l'eccezione se interrotto*

## Specifica (parziale) della classe Thread

---

**public final void stop () throws ThreadDeath**

*/\* causa la terminazione del thread sollevando  
l'eccezione ThreadDeath; se il thread era sospeso  
viene riesumato; se dormiva viene svegliato; se  
non era neanche iniziato, l'eccezione viene  
sollevata appena si fa lo start();  
l'eccezione può essere catturata e gestita ma  
deve comunque essere rimbalzata al metodo  
chiamante per far terminare correttamente il  
thread \*/*

## Specifica (parziale) della classe Thread

---

```
public final void suspend ()
```

```
/* viene sospeso; se lo è già non fa niente */
```

```
public final void resume ()
```

```
/*viene riesumato; se non era sospeso non fa nulla */
```

## Creazione di threads: stile 1

---

- **definiamo una sottoclasse di Thread che ridefinisce il metodo run()**
  - **il metodo contiene il codice che vogliamo eseguire nel thread**
  - **la sottoclasse non ha bisogno di avere un costruttore**
    - **all'atto della creazione di un nuovo oggetto si chiama automaticamente il costruttore Thread()**
  - **dopo aver creato l'oggetto della sottoclasse, il codice parte invocando il metodo start()**

## Un esempio di thread stupido 1

---

- cosa fa il metodo `run()` che contiene il codice che vogliamo eseguire nel thread
  - visualizza il thread corrente
  - stampa nell'ordine i numeri da 0 a 4, con un intervallo di 1 secondo
    - l'attesa viene realizzata con il metodo statico `sleep()` che deve essere incluso in un `try` perché può sollevare l'eccezione `InterruptedException` se interrotto da un altro thread
  - visualizza il messaggio "Fine run"

## Un esempio di thread stupido 2

---

```
public void run(){
    System.out.println ("Thread run" +
        Thread.currentThread ( ));
    for (int i = 0; i < 5; i++) {
        System.out.println (i);
        try {Thread.currentThread ( ).sleep (1000); }
        catch (InterruptedException e) {    } }
    System.out.println ("Fine run");}
```

## Creazione di threads stile 1: esempio il thread

- la sottoclasse di Thread

```
public class MioThread extends Thread {  
    public void run(){  
        System.out.println ("Thread run" +  
            Thread.currentThread ());  
        for (int i = 0; i < 5; i++) {  
            System.out.println (i);  
            try {Thread.currentThread ().sleep (1000); }  
            catch (InterruptedException e) { } }  
        System.out.println ("Fine run"); } }  
}
```

## run()

- Il programma non invoca run() direttamente
- Se si vuole avviare un Thread t, si può chiamare t.start() e la JVM si occuperà di chiamare t.run().
- start() porta a conoscenza della JVM che il codice di questo metodo ha la necessità di essere eseguito su un proprio thread.

```
Thread t = new Thread(this);  
t.run();
```

← Errore

```
Thread t = new Thread(this);  
t.start();
```

← Ok



## Creazione di threads stile 1: esempio il programma “principale”

---

1. visualizza il thread corrente
2. crea e manda in esecuzione il thread
3. fa dormire il thread corrente per 2 secondi
4. visualizza il messaggio “Fine main”
5. termina

```
public class ProvaThread {  
    public static void main (String argv[ ]) {  
        System.out.println ("Thread corrente: " +  
            Thread.currentThread ());  
        MioThread t = new MioThread ();  
        t.start ();  
        try { Thread.sleep (2000); }  
        catch (InterruptedException e) { }  
        System.out.println ("Fine main"); } }
```

## Creazione di threads stile 1: esempio il risultato

---

```
Thread corrente: Thread[main,5,system]  
Thread run: Thread[Thread-0,5,system]  
0  
1  
Fine main  
2  
3  
4  
Fine run
```

## Creazione di threads: stile 2

---

- **definiamo una classe *c* che implementa l'interfaccia *Runnable***
  - **nella classe deve essere definito il metodo *run()***
  - **non è necessario che siano definiti costruttori**
- **dopo aver creato un oggetto di tipo *c*, creiamo un nuovo oggetto di tipo *Thread* passando come argomento al costruttore *Thread(Runnable t)* l'oggetto di tipo *c***
- **il thread (codice del metodo *run()* di *c*) viene attivato eseguendo il metodo *start()* dell'oggetto di tipo *Thread***

## L'INTERFACCIA *Runnable*

---

- Il metodo *run()* dei *Thread* è anche dichiarato nella interfaccia *Runnable*. Quindi, come metodo alternativo per definire dei thread si può:
  - definire una propria classe che *implementi l'interfaccia *Runnable**, definendo il metodo *run()*.
- *Questa via è più flessibile*, (rispetto all'ereditare dalla classe *Thread*) in quanto consente di definire classi di thread che non siano per forza sottoclassi di *Thread* ma che siano sottoclassi di altre classi generiche.

- 
- ❑ E' il discorso che era stato fatto nella ereditarietà multipla....vogliamo definire delle classi che ereditino da altre classi, quelle che servono nello specifico problema, e poi fare in modo che questa classe abbia anche le proprietà dei threads.
  - ❑ Poiché non possiamo ereditare sia da Threads che dalla classe di interesse, usiamo il concetto di interfaccia:
    - ereditiamo dalla classe di interesse
    - poi implementiamo la interfaccia runnable
  - ❑ Per creare e usare un thread quindi bisogna:
    - *prima* creare una istanza della propria classe (che implementa l'interfaccia Runnable)
    - *poi* creare una istanza di Thread a partire da tale oggetto(incapsulandovi tale oggetto).

## Runnable interface

---

- ❑ L'interfaccia Runnable contiene solo un metodo:
  - `public void run()`
  - Questo metodo è chiamato per essere eseguito in un thread separato quando viene chiamato il metodo `start()` del Thread.
  - Questo metodo è *pure callback*.
    - ❑ Chiamando questo metodo direttamente non si inizializza un nuovo thread

## Creazione di threads stile 2: esempio

---

```
public class Prova implements Runnable {  
    public static void main (String argv[ ]) {  
        System.out.println ("Thread corrente: " +  
            Thread.currentThread ( ));  
        Prova pt = new Prova ( );  
        Thread t = new Thread(pt);  
        t.start ( );  
        try { Thread.sleep (2000); }  
        catch (InterruptedException e) { }  
        System.out.println ("Fine main"); }  
}
```

## Creazione di threads stile 2: esempio

---

```
public void run(){  
    System.out.println ("Thread run" +  
        Thread.currentThread ( ));  
    for (int i = 0; i < 5; i++) {  
        System.out.println (i);  
        try {Thread.currentThread ( ).sleep (1000); }  
        catch (InterruptedException e) { } }  
    System.out.println ("Fine run"); } }
```

## CONTROLLO DEI THREAD

- Un thread continua la propria esecuzione o fino alla sua terminazione naturale, o fino a che o fino a che non viene *espressamente fermato* da un altro thread che invochi su di esso il metodo stop()
- un thread può anche essere *sospeso* temporaneamente dalla sua esecuzione
  - perché un operazione blocca l'esecuzione, come una operazione di input che implica di attendere dei dati. Il Thread si blocca finché i dati non sono disponibili.
  - perché invoca la funzione Thread.sleep();
  - da un altro thread che invochi su di esso il metodo suspend(), per *riprendere* poi la propria esecuzione quando qualcuno invoca su di esso il metodo resume().

## stop, suspend, resume: PROBLEMI

- **ATTENZIONE!** stop(), suspend() e resume() **sono deprecati in Java2, in quanto non sicuri.**
- Questi tre metodi *agiscono in modo brutale* su un thread (che non può “difendersi” in nessun modo), e così facendo **rischiano di creare situazioni di blocco (deadlock)**:
  - se il thread sospeso / fermato stava facendo un'operazione critica, essa *rimane a metà* e l'applicazione viene a trovarsi in uno stato *inconsistente*
  - se il thread sospeso / fermato aveva acquisito una risorsa, essa *rimane bloccata da quel thread* e nessun altro può conquistarla
  - se a questo punto un altro thread si pone in attesa di tale risorsa, si crea una situazione di deadlock.

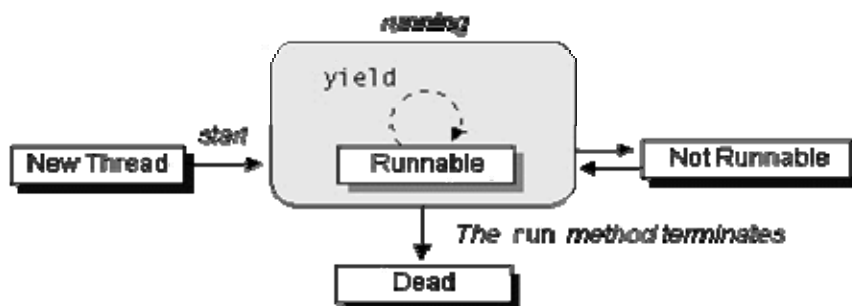
## stop, suspend, resume: PROBLEMI

### □ LA SOLUZIONE

- La soluzione suggerita da Java 2 consiste nell'adottare un *diverso approccio*:
  - *non* agire d'imperio su un thread,
  - ma *segnalargli l'opportunità* di sospendersi / terminare,
  - **lasciando però al thread il compito di *sospendersi / terminare*,**
  - in modo che possa farlo in un momento “non critico”.
  - la funzione ***yeld()*** permette a un thread di cedere l'esecuzione ad un altro!

## Vita di un thread

- Il seguente diagramma mostra gli stati in cui si può trovare un thread Java durante la sua vita.
- Illustra anche quali metodi possono causare una transizione da uno stato ad un'altro.



## Vita di un thread

---

```
Thread t = new Thread(this);    // New Thread  
t.start();
```

- Quando un thread è creato (New Thread) nel suo stato iniziale è un empty Thread object
  - Nessuna risorsa di sistema è allocata per esso.
- Quando un thread è in questo stato, su di esso può essere invocato il metodo start per avviare il.
  - Chiamando il metodo start su un oggetto thread che non si trova in tale stato iniziale si causa l'eccezione `IllegalThreadStateException`.

## Vita di un thread

---

```
Thread t = new Thread(this);    // New Thread  
t.start(); // starting a thread
```

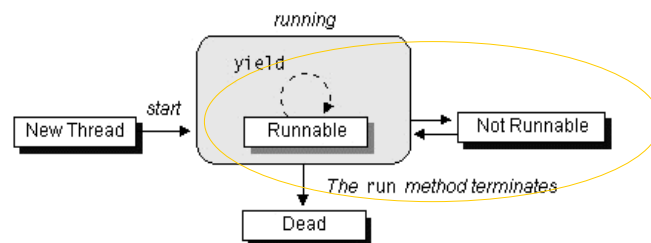
- Il metodo `start` :
  - Crea le risorse di sistema necessarie per eseguire il thread,
  - Schedula il thread per eseguirlo,
  - E chiama il metodo della classe thread chiamato `run`
- Dopo il return del metodo `start` il thread è "running" (in esecuzione).

## Vita di un thread

Un thread che è stato inizializzato è nello stato Runnable.

Molti computer hanno un singolo processore, quindi è impossibile eseguire tutti i "running" threads allo stesso tempo.

- Il Java runtime system deve implementare uno schema di scheduling scheme che permette la condivisione del processore fra tutti i "running" threads.
- Così, ad un dato istante un solo "running" thread è realmente in esecuzione, gli altri sono in waiting per il loro turno di CPU.



## Vita di un thread

- Un thread diventa Not Runnable (o **blocked**) quando uno di questi eventi ha luogo
- È invocato il suo metodo sleep.
- Il thread chiama il metodo wait per aspettare che una specifica condizione sia soddisfatta
- Il thread è bloccato su un I/O.
- Per ogni entrata in un Not Runnable state, c'è una specifica e distinta azione che ritorna il thread nello stato di Runnable.
  - Se un thread è stato posto in sleep, il numero specifico di milliseconds deve trascorrere.
  - Se un thread è in attesa per una condizione, allora un altro oggetto deve notificare il thread in attesa di un cambio nelle condizioni chiamando notify o notifyAll
  - Se un thread è blocked su I/O, allora l'I/O deve essere completato



## Stopping di un thread

- Un programma non blocca (stop) un thread attraverso la chiamata di un metodo
- Piuttosto, un thread termina naturalmente.
- Per esempio, il ciclo for nel metodo run è un loop finito, esso terminerà dopo 100:

```
public void run() {  
    for (int i = 0; i < 100; i++) {  
        System.out.println("i = " + i);  
    }  
}
```

- Un thread con il metodo run muore naturalmente quando completa il ciclo e il metodo run termina (exit)

## Stato di un Thread

- La classe Thread ha un metodo chiamato `isAlive()`.
- Il metodo `isAlive` ritorna `true` se il thread è stato started e non stopped.
- Se il metodo `isAlive` ritorna **false**, il thread è:
  - Un New Thread
  - O è morto.
- Se il metodo `isAlive` ritorna **true**, esso è :
  - Runnable
  - or Not Runnable.
- Non è possibile distinguere un New Thread da un Dead thread, ne è possibile distinguere un Runnable thread da un Runnable thread.

## Thread Scheduling

---

- ▣ Noi abbiamo detto che i threads sono eseguiti concorrentemente. Tale concorrenza è vera, di norma, solo concettualmente
- ▣ La maggior parte delle configurazioni di computer sono single CPU, così i threads realmente sono eseguiti uno alla volta in modo da fornire una illusione di concorrenza..
- ▣ L'esecuzione di threads multipli su un single CPU, in qualche ordine, è detta *scheduling*.
- ▣ Il Java runtime supporta un semplice algoritmo di scheduling deterministico conosciuto come *fixed priority scheduling*.
  - Questo algoritmo schedula i threads sulla base delle loro priorità relative

## Thread Scheduling

---

- ▣ Quando un thread Java viene creato esso eredita la sua priorità dal thread che lo ha creato
  - ▣ È possibile modificare la priorità di un thread in qualsiasi momento dopo la sua creazione utilizzando il metodo `setPriority`.
  - ▣ Le priorità dei Thread sono interi compresi fra `MIN_PRIORITY` e `MAX_PRIORITY` (costanti definite nella classe `Thread`).
- Il valore intero più grande corrisponde alla più alta priorità

## Thread Scheduling

---

- ▣ Ad un dato istante, quando più threads sono pronti per essere eseguiti, il runtime system sceglie il runnable thread con la più alta priorità per l'esecuzione.
- ▣ Solo quando il thread stops, o diventa not runnable per qualche ragione il thread di priorità minore viene eseguito
- ▣ Se due threads della stessa priorità sono in attesa per la CPU, lo scheduler esegue uno di loro seguendo un modello round-robin.
- ▣ Il thread scelto sarà eseguito fino a quando
  - Un thread con più alta priorità non diventa runnable.
  - Il suo metodo run esiste.
  - Su un sistema che supporta il time-slicing, il suo slice di tempo è terminato
- ▣ Quindi, il secondo thread ha una possibilità di essere eseguito, e così via.

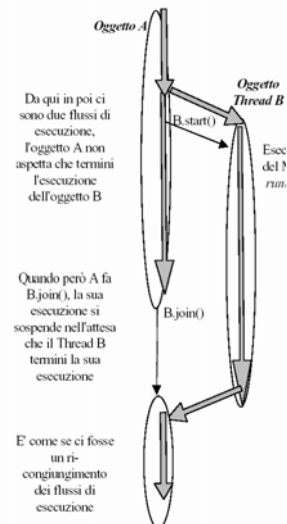
## Thread Scheduling

---

- ▣ Lo scheduler dei thread del Java runtime system è *preemptive*.
- ▣ Se in qualsiasi momento un thread con una più alta priorità di tutti gli altri runnable threads diventa runnable, allora il runtime system sceglie il nuovo thread di più alta priorità per l'esecuzione
- ▣ Il nuovo thread di più alta priorità viene detto che "*preempt*" gli altri threads.
- ▣ Ad ogni istante il thread di massima priorità dovrebbe essere in esecuzione. Comunque, ciò non è garantito.

## CONTROLLO DEI THREAD: JOIN

Si può decidere di aspettare che termini l'esecuzione di un thread.  
Per fare questo, bisogna invocare l'operazione `join()` sul thread che si intende aspettare



## Esempio

```
class Esempio4 {
    public static void main() {
        System.out.println("Main thread");
        while(true) {
            MyThread t = new MyThread();
            System.out.println("New thread " + t.getName() + " started");
            t.start();
            try {t.join(); // attende che t termini}
            catch (InterruptedException e) {}
        }
    }
}
```

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Qui thread " +
            Thread.currentThread().getName() );
    }
}
```

- il main è un ciclo infinito che crea continuamente nuovi thread (fino a che non viene chiuso con CTRL-C)
- prima di creare un nuovo thread, il main attende, tramite `join()`, che termini quello precedentemente in esecuzione
- l'attesa di `join()` potrebbe essere interrotta da un'eccezione
- necessità di try/catch

## Come rendere un programma multithreaded

---

- ▣ Come detto ci sono 2 modi per rendere un programma multithreaded...
  - Una classe implements Runnable e definisce il metodo run(). Questa classe viene passato nel costruttore di un Thread e iviene chiamato il metodo start() per eseguite il metodo run().
  - Estendere la classe Thread ed eseguire l'overriding del metodo run().

## Thread User e Thread Daemon

---

- ▣ Un *daemon* thread serve *user threads*.
- ▣ Un applicazione continua ad essere eseguita fino a quando almeno uno dei suoi threads è vivo.
- ▣ Quando un user thread termina in una applicazione, la the JVM cerca se ci sono altri user threads ancora vivi.
  - Se si, l'applicazione continua altrimenti l'applicazione termina perchè la JVM stessa termina.

## User e Daemon threads

### ■ Il seguente codice illustra daemon e user threads

```
class NotRunForever implements Runnable
{
    // il thread main svolge il ruolo di user o non di daemon
    public static void main (String [] args) {
        NotRunForever me = new NotRunForever();
        Thread t1 = new Thread(me);
        t1.setDaemon(true);      //t1 è il daemon
        t1.start();
        try {Thread.sleep(50);    }
        catch (InterruptedException e) {System.err.println(e);}
        System.out.println("il thread main termina");
    }
    public void run() {
        while (true) {
            System.out.println("il thread 2 è in esecuzione");
            try {Thread.sleep(10);}
            catch (InterruptedException e) {System.err.println(e);}
        }
    }
}
```

## User and Daemon threads

- Il thread main è il solo user thread dell'applicazione, così l'applicazione termina quando il main termina
- Un user thread costruisce altri user threads
- Un daemon thread costruisce altri daemon threads
- Il metodo setDaemon deve essere invocato prima che il thread sia started
- Il JVM's garbage collector viene eseguito come un daemon thread perché la JVM stessa dovrebbe terminare se nessun user thread è vivo.

## Thread groups

---

- ❑ Ogni thread Java è un membro di un *thread group*.
- ❑ Il Thread groups fornisce un meccanismo per collezionare multiple threads in un singolo oggetto e manipolare questi threads tutti insieme piuttosto che individualmente.
- ❑ Per esempio, è possibile eseguire una chiamata al metodo start o suspend di tutti i threads all'interno di un group con la chiamata di un singolo metodo
- ❑ Java thread groups sono implementati dalla classe `java.lang.ThreadGroup`.

## Thread groups

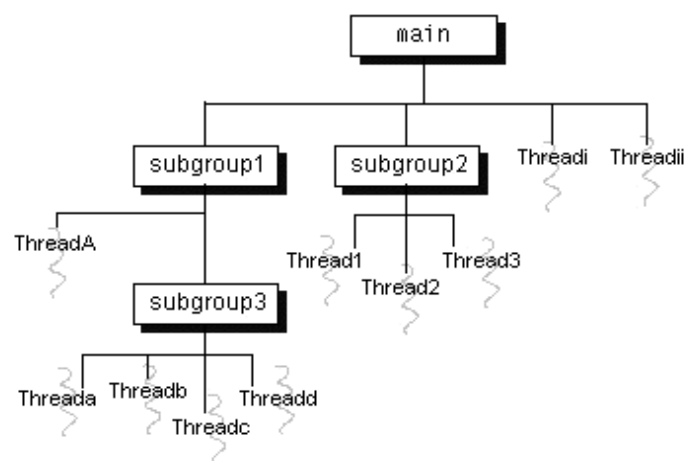
---

- ❑ Il runtime system pone un thread in un thread group durante la costruzione di un thread.
- ❑ Quando viene creato un thread, si può:
  - Permettere al runtime system di porre nuovi thread in un reasonable default group
  - O si può esplicitamente selezionare nuovi thread's group.
- ❑ Il thread è un membro permanente del gruppo in cui è inserito all'atto della creazione
  - Non è possibile spostare un thread in un nuovo gruppo dopo la sua creazione.

## Default Thread Group

- Se tu crei un nuovo Thread senza specificare il suo gruppo nel costruttore, il runtime system automaticamente pone il nuovo thread nello stesso gruppo (*current thread group*) così come il thread che lo ha creato (*current thread*)
- Quando una applicazione Java inizia il Java runtime system crea un ThreadGroup di nome main.
- Se non specificato altrimenti, tutti i nuovi threads creati diventano membri del that main thread group.
  - NOTA: se tu crei un thread all'interno di un applet, il nuovo thread's group può essere diverso dal main, a seconda del browser o viewer che esegue l'applet.

## Default Thread Group





## Specifica di un thread group

---

- ❑ Per inserire nuovi thread in un thread group bisogna specificarlo esplicitamente quando si creathread.
- ❑ La classe Thread ha tre costruttori che permettono di settare un nuovo gruppo di thread:

```
public Thread(ThreadGroup group, Runnable runnable)
public Thread(ThreadGroup group, String name)
public Thread(ThreadGroup group, Runnable runnable, String
name)
```

- ❑ crea un nuovo thread
- ❑ lo inizializza utilizzando I parametri

## Specifica di un thread group

---

- ❑ Il seguente codice crea un threadgroup (myThreadGroup e inserisce un thread (myThread) nel gruppo:

```
ThreadGroup myThreadGroup =
    new ThreadGroup("My Group of Threads");
Thread myThread = new Thread(myThreadGroup,
    "a thread for my group");
```

- ❑ Il ThreadGroup passato al costruttore di un Thread non deve essere necessariamente creato dal programmatore ma è possibile utilizzare gruppi creati da Java runtime system o gruppi gruppi creati dall'applicazione nella quale l'applet è in esecuzione.

## Quale è il threadgroup di un thread?

- ▣ Per conoscere il threadgroup di un thread esiste il metodo `getThreadGroup` method:

```
theGroup = myThread.getThreadGroup();
```

- ▣ Conosciuto il ThreadGroup, si possono reperire informazioni sul threadgroup

## Sincronizzazione di Thread

- ▣ Threads asincroni
  - Ogni thread contiene tutti i dati e i metodi necessari per l'esecuzione
  - I threads evolvono indipendentemente
    - ▣ Nessun problema per le velocità relative dei threads.
- ▣ Cosa accade quando esistono dati condivisi tra più threads?
  - Problema produttore/consumatore

## Sincronizzazione 1

---

- con il multithreading parti di uno stesso programma girano in modo concorrente
  - per lo più in modo indipendente
  - a volte è necessario che certe operazioni vengano eseguite in sequenza
    - quando due o più thread accedono contemporaneamente a variabili correlate oppure a una stessa risorsa del sistema, come un file, una stampante o una connessione di rete, i risultati possono essere imprevedibili
    - occorrono strumenti che permettano di eseguire certe sezioni di codice a non più di un thread alla volta (sincronizzazione)

## Sincronizzazione 2

---

- Java fornisce il meccanismo di sincronizzazione dei *mutex* (contrazione di mutual exclusion)
- un mutex è una risorsa del sistema che può essere posseduta da un solo thread alla volta
- ogni istanza di qualsiasi oggetto ha associato un mutex
- quando un thread esegue un metodo che è stato dichiarato sincronizzato mediante il modificatore `synchronized`
  - entra in possesso del mutex associato all'istanza
  - nessun altro metodo sincronizzato può essere eseguito su quell'istanza fintanto che il thread non ha terminato l'esecuzione del metodo

## Sincronizzazione: esempio 1

```
public class ProvaThread {
    public static void main (String argv[]) {
        ProvaThread pt = new ProvaThread ();
        Thread t = new Thread (pt);
        t.start ();
        pt.m2();
        public void m1 () {
            synchronized (pt) {
                ...
            }
        }
        void m2 () {
            ...
        }
    }
}
```

due metodi, m1 e m2, vengono invocati contemporaneamente da due threads su uno stesso oggetto pt  
m1 è dichiarato synchronized mentre m2 no  
il mutex associato a pt viene acquisito all'ingresso del metodo m1 non blocca l'esecuzione di m2 in quanto esso non tenta di acquisire il mutex

## Sincronizzazione: esempio 2

```
public class ProvaThread2 implements Runnable {
    public static void main (String argv[]) {
        ProvaThread2 pt = new ProvaThread2 ();
        Thread t = new Thread(pt);
        t.start ();
        pt.m2(); }
    public void run(){ m1();}
    synchronized void m1 () {
        for (char c = 'A'; c < 'F'; c++) {
            System.out.println (c);
            try { Thread.sleep (1000); }
            catch (InterruptedException e) { } } }
    void m2 () {
        for (char c = '1'; c < '6'; c++) {
            System.out.println (c);
            try {Thread.sleep (1000); }
            catch (InterruptedException e) { } } }
}
```

## Sincronizzazione esempio: risultati

---

1  
A  
2  
B  
3  
C  
4  
D  
5  
E

## Sincronizzazione: esempio 3

---

- se si dichiara synchronized anche il metodo `m2`, si hanno due threads che tentano di acquisire lo stesso mutex
  - i due metodi vengono eseguiti in sequenza, producendo il risultato

1  
2  
3  
4  
5  
A  
B  
C  
D  
E

## Sincronizzazione: esempio 4

- il mutex è associato all'istanza
  - se due threads invocano lo stesso metodo sincronizzato su due istanze diverse, essi vengono eseguiti contemporaneamente

```
public class ProvaThread3 implements Runnable {  
    public static void main (String argv[ ]) {  
        ProvaThread3 pt = new ProvaThread3 ();  
        ProvaThread3 pt2 = new ProvaThread3 ();  
        Thread t = new Thread(pt);  
        t.start ();  
        pt2.m1(); }  
    public void run(){ m1();}  
    synchronized void m1 () {  
        for (char c = 'A'; c < 'F'; c++) {  
            System.out.println (c);  
            try { Thread.sleep (1000); }  
            catch (InterruptedException e) { } } } }
```

## Sincronizzazione esempio: risultati

```
A  
A  
B  
B  
C  
C  
D  
D  
E  
E
```

## Sincronizzazione di metodi statici

- ❑ anche i metodi statici possono essere dichiarati sincronizzati
  - poiché essi non sono legati ad alcuna istanza, viene acquisito il mutex associato all'istanza della classe `Class` che descrive la 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 contemporaneamente

## Sincronizzazione con metodi statici: esempio 1

```
public class ProvaThread4 implements Runnable {
    public static void main (String argv[ ]) {
        ProvaThread4 pt = new ProvaThread4 ( );
        Thread t = new Thread(pt);
        t.start ( );
        m2(); }
    public void run(){ m1();}
    synchronized void m1 ( ) {
        for (char c = 'A'; c < 'F'; c++) {
            System.out.println (c);
            try { Thread.sleep (1000); }
            catch (InterruptedException e) { } } }
    static synchronized void m2 ( ) {
        for (char c = '1'; c < '6'; c++) {
            System.out.println (c);
            try {Thread.sleep (1000); }
            catch (InterruptedException e) { } } }
```

## Sincronizzazione con metodi statici: esempio 2

---

### □ il risultato

1  
A  
2  
B  
3  
C  
4  
D  
5  
E

## 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

```
synchronized (istanza) {  
    comando1;  
    ...  
    comandon;}  

```

- la gestione di programmi multithread è semplificata poiché il programmatore non ha la preoccupazione di rilasciare il mutex ogni volta che un metodo termina normalmente o a causa di una eccezione, in quanto questa operazione viene eseguita automaticamente



## Sincronizzazione implicita: esempio

```
public class ProvaThread5 implements Runnable {  
    public static void main (String argv[]) {  
        ProvaThread5 pt = new ProvaThread5 ();  
        Thread t= new Thread(pt);  
        t.start ();  
        synchronized (pt) { pt.m2(); }  
        public void run(){ m1();}
```

```
synchronized void m1 () {  
    for (char c = 'A'; c < 'F'; c++) {  
        System.out.println (c);  
        try { Thread.sleep (1000); }  
        catch (InterruptedException e) { } } }
```

```
void m2 () {  
    for (char c = '1'; c < '6'; c++) {  
        System.out.println (c);  
        try { Thread.sleep (1000); }  
        catch (InterruptedException e) { } } }
```

- ▣ sequenzializza l'esecuzione dei due metodi anche se m2 non è sincronizzato

## Comunicazione fra threads

- ❑ la sincronizzazione permette di evitare l'esecuzione contemporanea di parti di codice delicate
  - evitando comportamenti imprevedibili
- ❑ il multithreading può essere sfruttato al meglio solo se i vari threads possono comunicare per cooperare al raggiungimento di un fine comune
  - esempio classico: la relazione produttore-consumatore
    - ❑ il thread consumatore deve attendere che i dati da utilizzare vengano prodotti
    - ❑ il thread produttore deve essere sicuro che il consumatore sia pronto a ricevere per evitare perdita di dati
- ❑ Java fornisce metodi della classe `Object`
  - disponibili in istanze di qualunque classe
  - invocabili solo da metodi sincronizzati

## Locking un Object

- ❑ Il segmento di codice entro un programma che accede ad un oggetto da diversi threads concorrenti sono chiamate "regioni critiche".
- ❑ Regioni critiche
  - Possono essere un blocco o un metodo
  - Sono identificate dalla keyword `synchronized`.
- ❑ La piattaforma Java associa un lock ad ogni oggetto che ha del codice `synchronized`.

## Metodi di Object per la comunicazione fra threads 1

---

- `public final void wait()`
  - il thread che invoca questo metodo rilascia il mutex associato all'istanza e viene sospeso fintanto che non viene risvegliato da un altro thread che acquisisce lo stesso mutex e 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

## Metodi di Object per la comunicazione fra threads 2

---

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

## Un esempio di comunicazione fra threads

- la classe `Monitor` definisce oggetti che permettono la comunicazione fra un thread produttore ed un thread consumatore
- gli oggetti di tipo `Monitor` possono
  - ricevere una sequenza di stringhe dal thread produttore tramite il metodo `send`
  - ricevere un segnale di fine messaggi dal produttore tramite il metodo `finemessaggi`
  - inviare le stringhe nello stesso ordine al thread consumatore tramite il metodo `receive`
  - inviare un segnale di fine comunicazione al consumatore tramite il metodo `finecomunicazione`
  - tutti i metodi di `Monitor` sono sincronizzati

## Specifica della classe `Monitor`

```
class Monitor {  
    /* un Monitor è un oggetto che può contenere un messaggio  
       (stringa) e che permette di trasferire una sequenza di  
       messaggi in modo sincrono da un thread produttore ad un  
       thread consumatore*/  
    synchronized void send (String msg)  
        /*se è vuoto, riceve msg e diventa pieno; altrimenti il thread  
           viene sospeso finché this non diventa vuoto*/  
    synchronized String receive ( )  
        /*se è pieno, restituisce l'ultimo messaggio ricevuto e diventa  
           vuoto; altrimenti il thread viene sospeso finché this non  
           diventa pieno*/  
}
```

## Specifica della classe Monitor

---

```
synchronized void finemessaggi ( )  
    /* chiude la comunicazione con il produttore  
       il thread produttore non può invocare altri metodi dopo  
       questo*/  
synchronized boolean finecomunicazione ( )  
    /* restituisce true se this è vuoto ed ha chiuso la comunicazione  
       con il produttore  
}
```

## Implementazione della classe Monitor 1

---

```
class Monitor {  
    /* un Monitor è un oggetto che può contenere un  
       messaggio (stringa) e che permette di trasferire  
       una sequenza di messaggi in modo sincrono da un  
       thread produttore ad un thread consumatore*/  
    private boolean pieno = false;  
    private boolean stop = false;  
    private String buffer;
```

## Implementazione della classe Monitor 1

---

```
synchronized void send (String msg) {  
    if (pieno) try { wait ( ); } catch (InterruptedException e) { }  
    pieno = true;  
    notify ( );  
    buffer = msg; }  
  
synchronized void finemessaggi ( ) {  
    stop = true; }
```

## Implementazione della classe Monitor 2

---

```
synchronized String receive ( ) {  
    if (!pieno)  
        try { wait ( ); }  
        catch (InterruptedException e) { }  
    pieno = false;  
    notify ( );  
    return buffer; }  
  
synchronized boolean finecomunicazione ( ) {  
    return stop & !pieno ; }  
}
```

## Un thread consumatore

---

- ❑ la classe Consumatore fa partire un thread che si occupa di visualizzare i dati (stringhe) prodotti da un thread produttore
  - il costruttore
    - ❑ riceve e memorizza in una variabile di istanza l'oggetto di tipo Monitor che si occupa di sincronizzare le operazioni tra produttore e consumatore
    - ❑ crea un nuovo thread
  - il metodo run
    - ❑ esegue un ciclo all'interno del quale acquisisce un messaggio dal monitor e lo stampa, finché la comunicazione non viene fatta terminare dal produttore

## Il thread consumatore

---

```
class Consumatore implements java.lang.Runnable    {
    Monitor monitor;
    Consumatore (Monitor m) {
        monitor = m;
        Thread t = new Thread (this);
        t.start ( ); }
    public void run () {
        while (! monitor.finecomunicazione() )
            System.out.println (monitor.receive ( ) );
        return; } }
```

## Un thread produttore

---

- ❑ la classe Produttore fa partire un thread che genera una sequenza finita di messaggi (stringhe)
  - il costruttore
    - ❑ riceve e memorizza in una variabile di istanza l'oggetto di tipo Monitor che si occupa di sincronizzare le operazioni tra produttore e consumatore
    - ❑ crea il nuovo thread
  - il metodo run
    - ❑ manda al Monitor uno dopo l'altro le stringhe contenute in un array e poi segnala la fine della comunicazione

## Come parte il tutto

---

- ```
public class Provaprodcons {  
    public static void main (String argv []) {  
        Monitor monitor = new Monitor();  
        Consumatore c = new Consumatore(monitor);  
        Produttore p = new Produttore(monitor); } }
```
- ❑ si creano i due threads ed il monitor per farli comunicare
    - c'è anche il thread del main che ritorna dopo aver fatto partire gli altri
  - ❑ la sincronizzazione e la comunicazione sono completamente contenute nella classe Monitor



## Come si sposa la concorrenza con l'astrazione via specifica

---

- ❑ incapsulando sincronizzazione e comunicazione in classi come **Monitor** possiamo
  - specificare astrazioni sui dati orientate alla gestione della concorrenza
    - ❑ con invarianti di rappresentazione
  - dimostrare che la loro implementazione soddisfa la specifica
    - ❑ ma non è sempre facile capire cos'è la funzione di astrazione
  - dimostrare proprietà dei programmi che li usano (inclusi i threads) usando solo le loro specifiche
    - ❑ quasi come se non ci fosse concorrenza
- ❑ in Java si possono fare programmi concorrenti in molti altri modi

## Come si sposa la concorrenza con il polimorfismo

---

- ❑ è immediato realizzare monitors parametrici rispetto al tipo dei messaggi scambiati
  - sia usando messaggi di tipo **Object**
  - che usando sottotipi di interfacce opportune

## Come si sposa la concorrenza con le gerarchie di tipo e l'ereditarietà

---

- ❑ molto male (inheritance anomaly)
  - è molto difficile riuscire ad ereditare metodi sincronizzati
  - è difficile applicare il principio di sostituzione

## Esempi di produttori/consumatori

---

- ❑ Una applicazione Java ha un thread (produttore) che scrive dati su un file mentre un secondo thread (consumatore) legge i dati dallo stesso file.
- ❑ Digitando caratteri sulla tastiera il thread produttore pone l'evento key in una coda di eventi e il thread consumatore consumer legge gli eventi dalla stessa coda..
- ❑ Ambedue questi esempi coinvolgono thread concorrenti che condividono una risorsa comune:
  - File
  - Coda di eventi.
- ❑ Pertanto i thread devono essere sincronizzati

---

```
public class Box {  
    private int contents;  
    public void put(int x) {contents = x;}  
    public int get() {return contents;}  
}
```

## Locking l'oggetto Box

---

```
public class Box {  
    private int contents;  
    private boolean available = false;  
    public synchronized int get() { ... }  
    public synchronized void put(int value) { ... }  
}
```

- Consumer non dovrebbe accedere a Box quando Producer sta cambiando il valore in Box
- Producer non eseguire modifiche quando Consumer sta recuperando il valore.
- Whenever control enters a synchronized method, the thread that called the method locks the object whose method has been called.
- Other threads cannot call a `synchronized` method on the same object until the object is unlocked.

## Locking

---

```
public synchronized void put(int value) {  
    // Box locked dal Producer ..  
    // Box unlocked dal Producer  
}
```

```
public synchronized int get() {  
    // Box locked dal Consumer ...  
    // Box unlocked dal Consumer  
}
```

- ▣ L'acquisizione e il rilascio di un lock è fatto automaticamente dal Java runtime system.
  - Nessna condizione race conditions
  - data integrity.

## Uso di notify e wait

---

```
public synchronized void put(int value) {  
    if (available == false) {  
        available = true;  
        contents = value;}  
}  
  
public synchronized int get() {  
    if (available == false) {  
        available = true;  
        return value;}  
}
```

## Using notify and wait (2)

- Metodo `get`.
  - Quando `Producer` non ha posto qualcosa in `Box` e `available` non è vero? (`get` non fa nulla.)
  - Allo stesso modo, se `Producer` chiama `put` prima che `Consumer` abbia prelevato il valore ... (`put` non fa nulla.)
- `Consumer` deve attendere fino a che `Producer` mette qualcosa in `Box` e il `Producer` deve notificare al `Consumer` che lo ha fatto.
- Allo stesso modo, `Producer` deve aspettare fino a quando `Consumer` non ha prelevato il valore (e notifica al `Producer` di averlo fatto) prima di riporre un altro valore in `Box`
- The two threads must coordinate more fully and can use `Object`'s `wait` and `notifyAll` methods to do so.

## Using notify and wait (3)

```
public synchronized int get() {  
    while (available == false) {  
        try {  
            // wait f  
            wait();  
        } catch (InterruptedException e) {  
        }  
    }  
    available = false;  
    // notify Producer that value has been retrieved  
    notifyAll();  
    return contents;  
}
```

Il codice del metodo `get` loops fino a che `Producer` non ha prodotto il nuovo valore.

Quando `Producer` mette qualcosa in `Box` esso notifica al `Consumer` chiamando il metodo `notifyAll`

## Using notify and wait (5)

---

```
public synchronized void put(int value) {
    while (available == true) {
        try {
            // wait for Consumer to get value
            wait();
        } catch (InterruptedException e) {
        }
    }
    contents = value;
    available = true;
    // notify Consumer that value has been set
    notifyAll();
}
```

## Producer/Consumer...

---

```
import java.util.*;
public class Test{
    public static void main(String s[]) {
        LinkedList list = new LinkedList();
        Producer p = new Producer(list,1);
        Consumer c = new Consumer(list, 1);
        p.start();
        c.start();
    }
}
```

```

public class Producer extends Thread{
    private Box pozzo;
    private int number;
    public Producer(Box l, int number)    {
        pozzo = l;
        this.number = number;
    }
    public void run (){
        for (int i=0; i<10; i++){
            pozzo.put(i);
            System.out.println("Produzione del numero "
                               +this.number + "put" + i);
            try {sleep((int)(Math.random()*10));}
            catch (InterruptedException e){};
        }
    }
}

```

```

public class Consumer extends Thread
{
    private Box pozzo;
    private int number;
    public Consumer(Box l, int number)
    {
        pozzo = l;
        this.number = number;
    }
    public void run ()
    {
        int value = 0;
        for (int i=0; i<10; i++)
        {
            value=pozzo.get();
            System.out.println("Consumo del numero " +
this.number + "get" + value);
            try
            {
                sleep((int)(Math.random()*1000));}
            catch (InterruptedException e){};
        }
    }
}

```

## Produttore/consumatore

---

- ❑ Producer and Consumer in questo esempio condividono i dati di un oggetto comune `LinkedList`.
- ❑ Il problema è che il `Producer` potrebbe essere più lento o più rapido del `Consumer`
- ❑ È necessaria la sincronizzazione fra i due threads

## Il problema di sincronizzare Producer/Consumer

---

- ❑ L'attività di `Producer` e `Consumer` può essere sincronizzata in due modi:
  - I due threads non devono accedere simultaneamente all'oggetto `Box`
    - ❑ Serve un lock
  - I due threads devono coordinarsi. Il `Producer` deve indicare al `Consumer` che il valore è pronto e il `Consumer` deve avere un modo per notificare che il valore è stato prelevato.
    - ❑ La classe `Thread` ha una collezione di metodi (`wait`, `notify`, e `notifyAll`) che consentono ai threads di aspettare una condizione e notificare ad altri threads che la condizione è cambiata



## High level concurrency

---

- ▣ Nuove caratteristiche introdotte in java 5.0 (java.util.concurrent)
  - Lock
  - Executors
  - Concurrent Collections
  - Atomic variable

## Lock Objects

---

- ▣ gli oggetti lock funzionano in maniera simile ai lock impliciti usati dai metodi sincronizzati
  - solo un thread per volta può acquisire un lock
  - lock permettono di utilizzare wait/notify attraverso oggetti **Condition** associati
- ▣ il maggiore vantaggio è legato alla possibilità di "tornare indietro" se il lock non è disponibile (tryLock)

```
public boolean impendingBow(Friend bower) {  
    Boolean myLock = false;  
    Boolean yourLock = false;  
    try {  
        myLock = lock.tryLock();  
        yourLock = bower.lock.tryLock();  
    } finally {  
        if (! (myLock && yourLock)) {  
            if (myLock) { lock.unlock(); }  
            if (yourLock) { bower.lock.unlock(); }  
        }  
    }  
    return myLock && yourLock;  
}
```

## Executor

- ▣ è più efficiente separare la creazione e la gestione dei thread dal resto dell'applicazione
- ▣ Ogni oggetto che ha queste funzioni è un esecutore
  - ExecutorInterface
  - Thread Pools (implementazione degli esecutori)

## Executor Interface

---

- ▣ Sono definite tre interfacce

- Executor:
- ExecutorService
- ScheduledExecutorService

## Exceutor

---

- ▣ fornisce un singolo metodo che sostituisce lo start

`(new Thread(r)).start();`  
`e.execute(r)`

- ▣ il funzionamento di execute()

- è analogo al meccanismo tradizionale
- utilizza un worker (thread) esistente
- immette il thread in una coda di attesa per un worker

## ExecutorService

---

- ▣ aggiunge un nuovo metodo `submit()`
- ▣ `submit` accetta come parametro
  - `Runnable`
  - `Callable`, che permettono di restituire un tipo `Future` che permette di accedere al tipo realmente restituito e di gestire lo stato (sia di `Callable` che di `Runnable`)
- ▣ è possibile sottomettere collezioni di oggetti `Callable`
- ▣ fornisce strumenti per gestire l'interruzione del `Thread`

## ScheduledExecutorService

---

- ▣ questa interfaccia aggiunge il metodo
  - `schedule()`: permette di eseguire oggetti `Runnable` o `Callable`
  - `scheduleAtFixedRate`
  - `scheduleWithFixedDelay`

## Thread Pools

---

- ▣ L'utente struttura l'applicazione mediante un insieme di tasks.
  - Task = segmento di codice che può essere eseguito da un esecutore. Può essere definito come un oggetto di tipo Runnable
  - Thread = esecutore in grado di eseguire tasks.
- ▣ Uno stesso thread può essere utilizzato per eseguire diversi tasks, durante la sua esecuzione
- ▣ Thread Pool = Struttura dati la cui dimensione massima è prefissata, che contiene riferimenti ad un insieme di threads
  - I thread del pool possono essere utilizzati per eseguire i tasks sottomessi per l'esecuzione dall'utente al thread pool

## Thread pool

---

- ▣ L'utente
  - Definisce i tasks della applicazione
  - Crea un thread di pool e stabilisce una politica per la gestione dei threads del pool, la politica stabilisce
    - ▣ quando i threads del pool vengono attivati: (al momento della creazione del pool, on demand, in corrispondenza dell'arrivo di un nuovo task,...)
    - ▣ se e quando è opportuno terminare l'esecuzione di un thread (ad esempio se non c'è un numero sufficiente di tasks da eseguire)
  - Sottomette i tasks per l'esecuzione al thread pool.

## Thread pool

- ❑ L'applicazione sottomette un task T al gestore del thread pool
  - Il gestore sceglie un thread dal pool per l'esecuzione di T. Scelte possibili:
    - ❑ utilizzare un thread attivato in precedenza, ma inattivo al momento dell'arrivo del nuovo task
    - ❑ creare un nuovo thread, purchè non venga superata la dimensione massima del pool
    - ❑ memorizzare il task in una struttura dati, in attesa di eseguirlo
    - ❑ respingere la richiesta di esecuzione del task
- ❑ Il numero di threads attivi nel pool può variare dinamicamente

## Thread pool: motivazioni

- ❑ Tempo stimato per la creazione di un thread: qualche centinaio di microsecondi.
- ❑ Creazione di un alto numero di threads può non essere tollerabile per certe applicazioni
- ❑ Thread Pooling
  - diminuisce l'overhead dovuto alla creazione di un gran numero di threads. Lo stesso thread può essere riutilizzato per l'esecuzione di più di un task
  - permette una migliore strutturazione del codice dell'applicazione.
- ❑ Tutta la gestione dei threads può essere delegata al gestore del thread pool

## LIBRERIA JAVA.UTIL.CONCURRENT

---

- ❑ L'implementazione del thread pooling può:
  - Fino a J2SE 4 doveva essere realizzata a livello applicazione
  - J2SE 5.0 definisce la libreria `java.util.concurrent` che contiene metodi per
- ❑ Creare un thread pool e il gestore associato
- ❑ Definire la struttura dati utilizzata per la memorizzazione dei tasks in attesa
- ❑ Decidere specifiche politiche per la gestione del pool

## CREARE UN THREADPOOL EXECUTOR

---

JAVA 5 definisce

- ❑ Alcune interfacce che definiscono servizi generici di esecuzione...

```
public interface Executor {  
    public void execute (Runnable task); }  
public interface ExecutorService extends Executor {  
    ..... }  
}
```
- ❑ diversi esecutori che implementano il generico `ExecutorService` (`ThreadPoolExecutor`, `ScheduledThreadPoolExecutor`,...)
- ❑ la classe `Executors` che opera come una Factory in grado di generare oggetti di tipo `ExecutorService` con comportamento predefiniti.
- ❑ I tasks devono essere incapsulati in oggetti di tipo `Runnable` e passati a questi esecutori, mediante invocazione del metodo `execute( )`

## THREAD POOLING: ESEMPI

```
import java.util.concurrent.*;
public class liftoff implements Runnable{
    int countDown = 3; // Predefinito
    public liftoff(){ }
    public String status( ){
        return "#" + Thread.currentThread() + "(" +
            (countDown > 0 ? countDown: "Via!!!")+"),";
    }
    public void run( ) {
        while (countDown-- > 0){
            System.out.print(status());
            try{ Thread.sleep(100);}
            catch(InterruptedException e){ }
        }
    }
}
```

### Esempio 1

```
public class esecutori {
    public static void main(String[]args){
        ExecutorService exec = Executors.newCachedThreadPool();
        for (int i=0; i<2; i++){
            exec.execute(new liftoff());
        }
    }
}
```

newCachedThreadPool ( ) crea un pool in cui, quando viene sottomesso un task viene creato un nuovo thread se tutti i threads del pool sono occupati nell'esecuzione di altri tasks. viene riutilizzato un thread che ha terminato l'esecuzione di un task precedente, se disponibile. Se un thread rimane inutilizzato per 60 secondi, la sua esecuzione termina viene rimosso dalla cache.



## ESEMPIO 2: OUTPUT

```
import java.util.concurrent.*;
public class esecutori {
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for (int i=0; i<3; i++){
            exec.execute(new liftoff( ));
            try{Thread.sleep (10000);}
            catch(InterruptedException e) { }
        }
    }
}
```

- ▣ La sottomissione di tasks al pool viene distanziata di 10 secondi. In questo modo
- ▣ l'esecuzione precedente è terminata ed è possibile riutilizzare un thread
- ▣ attivato precedentemente

## THREAD POOLING: ESEMPIO 3

```
import java.util.concurrent.*;
public class esecutori {
    public static void main(String[] args){
        ExecutorService exec = Executors.newFixedThreadPool
        (2);
        for (int i=0; i<3; i++){
            exec.execute(new liftoff());
        }
    }
}
```

- ▣ newFixedThreadPool (int i) crea un pool in cui, quando viene sottomesso un task
  - Viene riutilizzato un thread del pool, se inattivo
  - Se tutti i threads sono occupati nell'esecuzione di altri tasks, il task viene inserito in una coda, gestita dall'ExecutorService e condivisa da tutti i tasks..

## THREAD POOL EXECUTORS

```
import java.util.concurrent.*;
public class ThreadPoolExecutor implements
    ExecutorService {
    public ThreadPoolExecutor (int core PoolSize,
                              int maximum PoolSize, long
    keepAliveTime, TimeUnit unit,
    BlockingQueue<Runnable> workqueue);
    .....}
```

- ▣ Crea un threadpool con il corrispondente esecutore

## CREARE UN THREAD POOL

```
public ThreadPoolExecutor (int core
    PoolSize,
    int maximunPoolSize, long keepAliveTime,
    TimeUnit unit, BlockingQueue<Runnable>
    workqueue);
```

- ▣ CorePoolSize, MaximumPoolSize, keepAliveTime controllano la gestione dei threads del pool
- ▣ Workqueue è una struttura dati necessaria per memorizzare gli eventuali tasks in attesa di esecuzione

## Concurrent Collections

---

- ❑ Le collezioni sono state estese inserendo nuovi tipi che garantiscono la sincronizzazione
  - BlockingQueue: una coda che entra in uno stato di blocco (o timeout) se si tenta di inserire un dato in una coda piena o leggere da una coda vuota
  - ConcurrentMap: rende atomiche alcune azioni permettendo di non usare metodi sincronizzati
  - ConcurrentNavigableMap: permette di effettuare confronti approssimativi
- ❑ Tutti queste collezioni permettono di evitare errori di consistenza nell'accesso alla memoria

## THREAD POOL: GESTIONE DINAMICA

---

- ❑ Core: dimensione minima del pool: il supporto crea un pool di dimensione
- ❑ Core.
  - È possibile allocare core threads al momento della creazione del pool mediante il metodo `prestartAllCoreThreads()`. I threads creati rimangono inattivi in attesa di tasks da eseguire
  - I threads vengono creati "on demand". Quando viene sottomesso un nuovo task, viene creato un nuovo thread, anche se alcuni dei threads già creati sono inattivi. L'obiettivo è di riempire il pool prima possibile
- ❑ `MaxPoolSize`: dimensione massima del pool

## THREAD POOL: GESTIONE DINAMICA

---

- ❑ Se sono in esecuzione tutti i core threads, un nuovo task sottomesso viene inserito in una coda C.
  - C deve essere una istanza di BlockingQueue
  - C viene passata al momento della costruzione del threadpool (ultimo parametro del costruttore)
  - E' possibile scegliere diversi tipi di coda (tipi derivati da BlockingQueue). Il tipo di coda scelto influisce sullo scheduling.
    - ❑ I task vengono poi prelevati da C e inviati ai threads che si rendono disponibili
    - ❑ Solo quando C risulta piena si crea un nuovo thread attivando così k threads,  $\text{core} \leq k \leq \text{MaxPoolSize}$

## THREAD POOL: GESTIONE DINAMICA

---

- ❑ Da questo punto in poi, quando viene sottomesso un nuovo task T
  - se esiste un thread TH inattivo T viene assegnato a TH
  - se non esistono threads inattivi, si preferisce sempre accodare un task piuttosto che creare un nuovo thread
  - solo se la coda è piena, si attivano nuovi threads
  - Se la coda è piena e sono attivi MaxPoolSize threads, il thread viene respinto e viene sollevata un'eccezione

- 
- Supponiamo che un thread TH termini l'esecuzione di un task, e che il pool contenga k threads
    - Se  $k \leq \text{core}$ : il thread si mette in attesa di nuovi tasks da eseguire.
    - L'attesa è indefinita.
    - Se  $k > \text{core}$ , si considera il timeout T definito al momento della costruzione del thread pool
      - se nessun thread viene sottomesso entro T, TH termina la sua esecuzione, riducendo così il numero di threads del pool
      - Timeout: occorre definire
        - un valore (es: 50000) e
        - l'unità di misura utilizzata (es: TimeUnit. MILLISECONDS)

## THREAD POOL: TIPI DI CODA

---

- SynchronousQueue: dimensione uguale a 0. Ogni nuovo task T
  - viene eseguito immediatamente oppure respinto.
  - T viene eseguito immediatamente se esiste un thread inattivo oppure se è possibile creare un nuovo thread (numero di threads  $\leq \text{MaxPoolSize}$ )
  - LinkedBlockingQueue: dimensione illimitata
    - E' sempre possibile accodare un nuovo task, nel caso in cui tutti i tasks attivi nell'esecuzione di altri tasks
    - La dimensione del pool di non può superare core
- ArrayBlockingQueue: dimensione limitata, stabilita dal programmatore

## ATTENDERE LA TERMINAZIONE DI UN THREAD: METODO JOIN

---

- ▣ Un thread J può invocare il metodo join( ) su un oggetto T di tipo thread
- ▣ J rimane sospeso sulla join( ) fino alla terminazione di T.
- ▣ Quando T termina, J riprende l'esecuzione con l'istruzione successiva alla join( ).
- ▣ Un thread sospeso su una join( ) può essere interrotto da un altro thread che invoca su di esso il metodo interrupt( ).
- ▣ Il metodo può essere utilizzato nel main per attendere la terminazione di tutti i threads attivati.

## Variabili atomiche

---

- ▣ sono definite alcune classi che supportano operazioni atomiche su singole variabili
- ▣ tutte le variabili forniscono i metodi get() e set() che eseguono le operazioni atomiche
- ▣ il metodo compareAndSet() permette di eseguire un'altra comune operazione

```
class AtomicCounter {  
    private AtomicInteger c = new AtomicInteger(0);  
    public void increment() { c.incrementAndGet(); }  
    public void decrement() { c.decrementAndGet(); }  
    public int value() { return c.get(); }  
}
```