

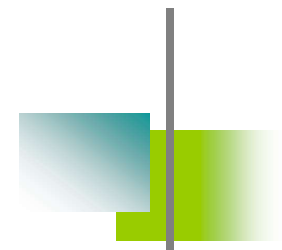


---

# Multithreading



**Dott. Romina Fiorenza**  
**[fiorenza.romina@gmail.com](mailto:fiorenza.romina@gmail.com)**





# Task e Multitasking

---

Un **processo** è un programma in esecuzione (o una parte di esso)

Il **multitasking** è la capacità del sistema operativo di eseguire **più processi** *contemporaneamente*

Lo **scheduler** è il componente del sistema operativo che si occupa di decidere come distribuire l'accesso alla CPU per eseguire i processi → come eseguire il **context switch**

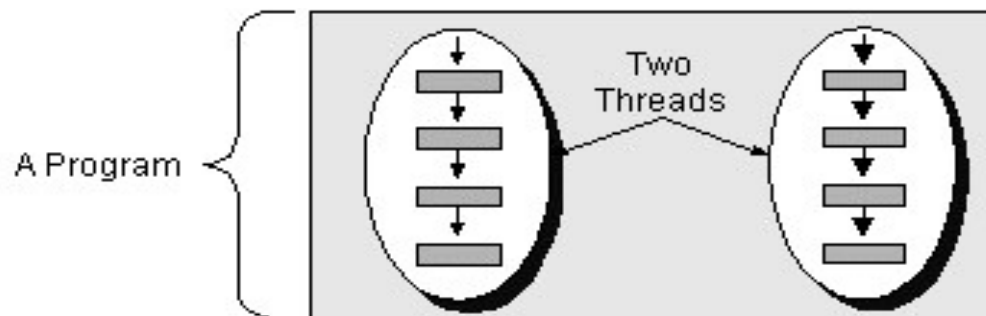
Caratteristiche del multitasking:

- ogni processo ha un **proprio spazio indirizzato**
- la comunicazione tra processi è **complessa** ed è a carico del **sistema operativo**

# Thread

Un **thread** è un flusso di esecuzione all'interno di un processo.

- Può essere dunque un processo o una parte di esso.
- Un processo si può quindi vedere come un insieme di flussi: ciascuno di essi è un **Thread**.
- Ogni processo ha almeno un thread **principale**, che può dare origine a quelli **secondari**
- Ogni thread si possono ordinare in base alla **priorità**



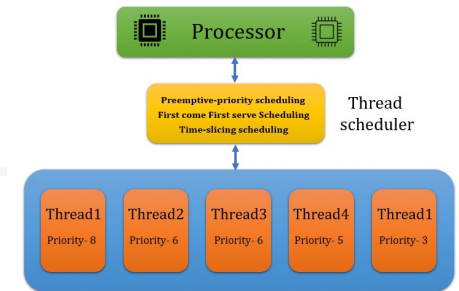


# Multithreading

Il **multithreading** rappresenta la possibilità di eseguire più thread contemporaneamente.

- Quindi piccole unità di calcolo relative ad uno stesso programma vengono eseguite in modo **quasi** simultaneo
- Risultato → aumentare la velocità di calcolo e l'efficienza
- Caratteristiche del multithreading:
  - Thread diversi girano nello **stesso spazio di memoria**
  - comunicazione efficiente (**memoria condivisa**) ma potenziali problemi di **sincronizzazione** (per l'accesso a oggetti condivisi)

# JVM e thread



Il criterio di schedulazione dei thread della JVM è il **preemptive - priority scheduling** che sfrutta la **priorità**, l'**ordine di arrivo** e il **time slicing**

- Ogni thread ha una priorità (valore tra 1 e 10) e un tempo di arrivo (istante di tempo in cui il thread risulta pronto all'esecuzione, cioè è Runnable)

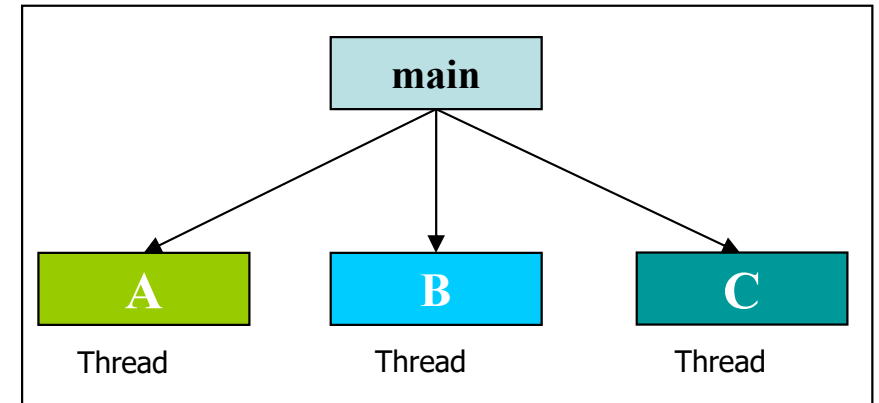
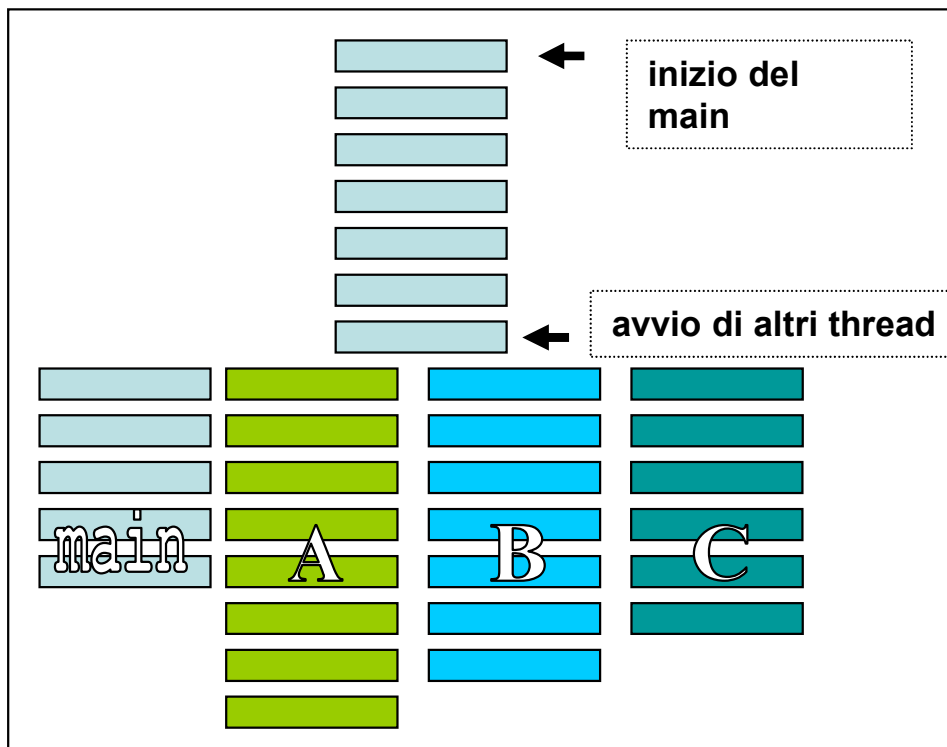
Come lavora lo scheduler Java?

1. Vengono schedulati tutti i thread runnable e viene stabilito un intervallo di tempo della CPU per ciascun thread (**time slicing**)
2. Il thread a priorità più alta viene mandato in esecuzione (**priority**)
3. Lo switch context può avvenire se
  1. sopraggiunge un thread a priorità maggiore oppure
  2. il thread è stato interrotto (o ha terminato il task) oppure
  3. lo slot di tempo assegnato si è esaurito
4. Se ci sono più thread con la stessa priorità vengono gestiti in una coda secondo il principio *First come, First serve* (**ordine arrivo**)

# Il thread main

Il metodo `main` gira anch'esso in un thread chiamato thread principale (main thread)

Dal main possiamo lanciare thread secondari



- I nuovi thread lavorano in parallelo tra loro e rispetto al main stesso
- Ognuno cerca di completare il proprio lavoro, interagendo eventualmente con gli altri.



# Classi di supporto

---

- Le API di base a supporto del multithreading sono tutte nel package `java.lang` e sono:
  - `interface Runnable`
  - `class Thread`
  - `class Object`
- `Runnable` definisce il funzionamento di un thread
- `Thread` fornisce metodi per creare e gestire thread
- `Object` fornisce i metodi per la sincronizzazione degli accessi ai thread



# Creare un thread

- Il comportamento di un thread è definito dall'interfaccia *Runnable*

```
public interface Runnable{  
    // rappresenta l'azione che il thread dovrà compiere  
    public void run();  
}
```

- La classe **Thread** è un'implementazione di *Runnable*
  - fornisce strumenti per gestione dei thread (**creazione inclusa**)
  - implementa `run()`, il quale però non esegue NULLA.
- **Creare un thread quindi comporta:**
  - **estendere** la classe Thread *oppure*
  - **implementare** l'interfaccia Runnable

NOTA: Runnable consente ad una classe di lavorare come un thread, quando questa non può estendere la classe Thread.





# Classe Thread

---

- Costruttori:

- Thread()
- Thread(Runnable target)
- Thread(Runnable target, String name)
- Thread(String name)

quando non specificato si ha

- `name = "Thread-"+n` che numera i thread partendo da zero (solo il main ha il suo nome come nome di default)
- `Runnable = null`

- Metodi:

- getName() torna il nome del thread
- setName(**String** name) imposta il nome del thread
- currentThread() **Static** torna il thread corrente

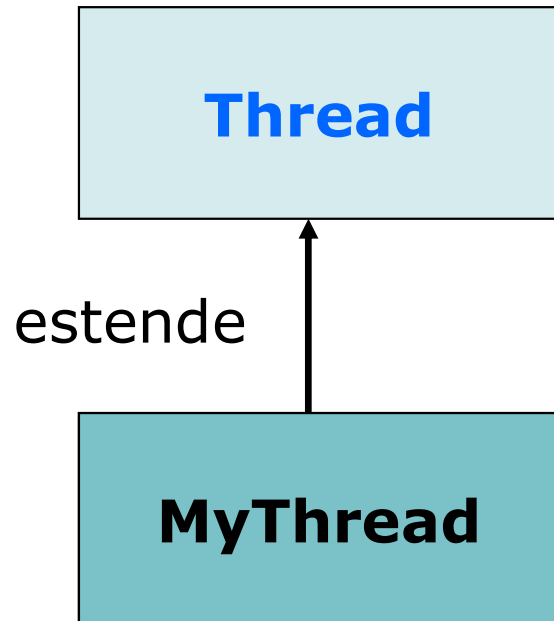


# Utilizzo di Thread

---

- **Passo 1:** estendere la classe `java.lang.Thread`
- **Passo 2:** ridefinire il metodo ***run()***
- **Passo 3:** creare il *thread* e avviarlo con ***start()***

# Schema



```
class MyThread extends Thread
{
    ...
    public void run() {...}
    ...
}
```

**Nel main**

```
...
MyThread tr = new MyThread();
tr.start();
```



# Avviare un thread

---

- Dopo la chiamata al costruttore, il Thread non è già un *thread of execution*, infatti bisogna avviarlo con il metodo **start()**
- Il metodo **Thread.start**
  - crea le risorse di sistema necessarie per eseguire il thread
  - schedula il thread per eseguirlo
  - chiama il metodo run della classe thread
  - Dopo il return di start il thread è **Runnable**
- Non bisognerebbe invocare direttamente **run()**, è il metodo start che inizializza il Thread e invoca poi il metodo operativo **run()**
- Omettendo la chiamata a **start()** il programma lavora in modo asincrono → No Multithreading!

# Esempio

- Omettendo la chiamata a **start()** il programma lavora in modo sequenziale → No Multithreading!

```
MyThread t = new MyThread ();  
// viene eseguito "tutto il metodo run"  
t.run();  
// a seguire viene eseguito il ciclo  
for(int i=0;i<200;i++)  
    System.out.println("ciclo - " + i);
```

questa chiamata verrà eseguita nel **corrente call stack** non in quello del thread appena creato

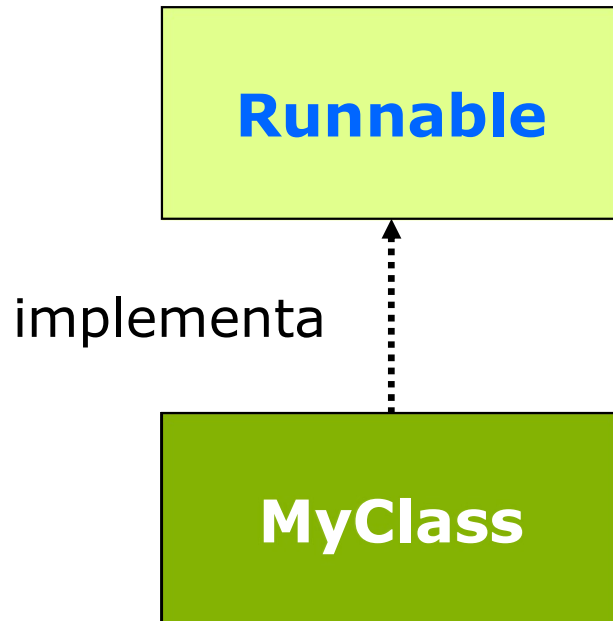
- NB: Si avviano SOLO i thread, gli oggetti Runnable NON si avviano!!! (non c'è il metodo)



# Utilizzo di Runnable

- **Passo 1:** implementare l'interfaccia ***Runnable*** di **`java.lang`** implementando il metodo ***run()***
- **Passo 2:** creare un oggetto ***Runnable***
- **Passo 3:** costruire un Thread attraverso il Runnable creato, utilizzando **`Thread(Runnable target)`**
- **Passo 4:** avviare il *thread* invocando ***start()*** sull'oggetto

# Schema



```
class MyClass implements Runnable {  
    ...  
    public void run() {...}  
    ...  
}
```

## Nel main

```
...  
MyClass p = new MyClass();  
Thread tr = new Thread(p);  
tr.start();
```



# La priorità

---

- La priorità di un Thread è un numero positivo tra 1 e 10
  - non è detto che la JVM riconosca esattamente 10 valori!
- La classe `java.lang.Thread` definisce
  - `public static final int MAX_PRIORITY` = 10
  - `public static final int MIN_PRIORITY` = 1
  - `public static final int NORM_PRIORITY` = 5
- Un thread nasce con priorità **NORMAL**
- La priorità di un thread si può leggere e modificare con i metodi
  - `getPriority()`
  - `setPriority(int)`
- La JVM non può cambiare la priorità di un Thread
  - Non è detto che un thread a priorità più alta inizi prima di uno a priorità più bassa

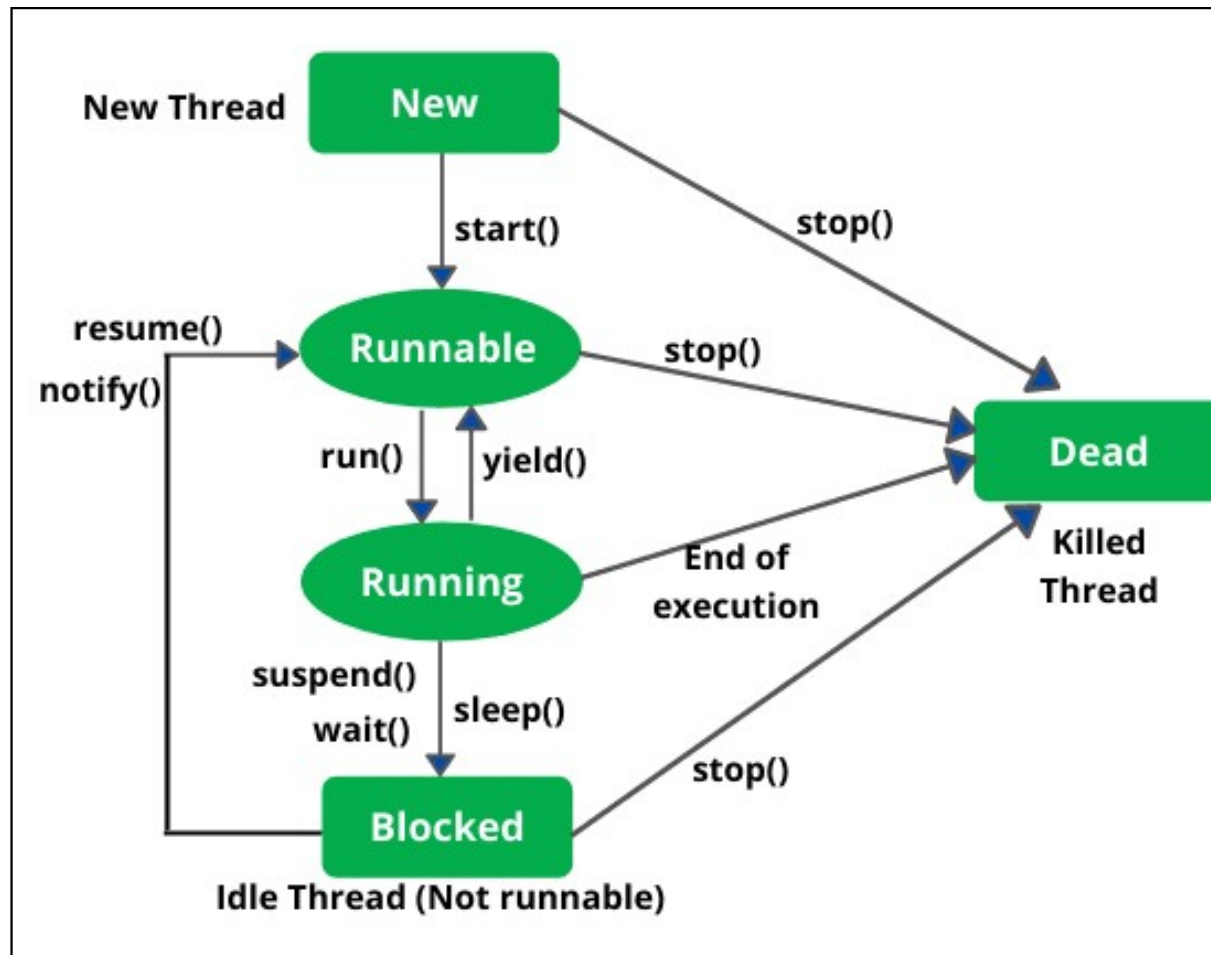




# User & daemon

- I Thread sono essenzialmente di 2 tipi:
  - Thread **user**
  - Thread **daemon**
- Tutti i thread nascono user, ma si possono impostare come daemon con il metodo:  
`void setDaemon(boolean)`  
DEVE essere invocato prima dello `start()`, col valore `true` imposta il Thread come daemon, viceversa resta user.
- Un daemon thread è un thread a bassa priorità, che esegue dei servizi in background ed è subordinato al thread user dentro il quale è stato avviato.
- **In particolare** → *Un Thread daemon **termina** quando il suo **Thread user termina***
- Thread user e thread daemon sono strutturalmente uguali, tranne che per la schedulazione e la chiusura.
- **IMPORTANTE:**  
*La JVM termina quando tutti i **thread user** sono terminati e non si cura del fatto che lo siano anche i **thread daemon***
- NB: fa eccezione il thread main. Se si crea un daemon dal main, ma esiste un altro thread user (diverso dal main), allora al termine del main, il thread daemon non termina finchè l'altro thread user non ha terminato.

# Ciclo di vita di un thread





# Passaggi di stato

- Un thread **eseguibile** oscilla continuamente tra Runnable e Running (in base all'utilizzo del processore)
- Un thread viene **sospeso/bloccato** se:
  - sono stati invocati `sleep`, `join`
  - ci sono operazioni di I/O bloccanti
  - è stato chiamato il metodo `suspend`\*
  - è stato chiamato il metodo `wait` (usato per gestione sincronizzazione)

Solo un thread in Running può passare in stato Waiting/Blocked  
Uscendo da questo stato si passa, in generale, in Runnable

- Un thread **muore**:
  - naturalmente, se il metodo `run` è terminato
  - “brutalmente”, se è stato chiamato lo `stop`\*
  - accidentalmente, se sono state sollevate eccezioni non gestite

\* **Deprecated** → evitare di invocarlo, optare per `interrupt()`

# Metodi bloccanti

- Si può *rallentare* il thread corrente (in Running), sospendendolo per un tempo prefissato, con il metodo

```
public static void sleep(long millisecond)
```



```
public static void sleep(long milli, int nanos)
```



- Si può *raccordare* il thread corrente con un thread target, invocando su quest'ultimo il metodo:

```
public final void join()
```

```
public final void join(long milli)
```



```
public final void join(long milli, int nanos)
```



In questo modo il thread corrente attende la fine dell'esecuzione del thread chiamante (eventualmente entro il tempo massimo specificato)



# Uso improprio di sleep

```
public class Example{  
    public static void main(String [] args) {  
        Thread one = new Thread();  
        one.start();  
        try {  
            one.sleep(5*1000);  
            // addormenta il thread corrente -> il main per 5 secondi  
        } catch (InterruptedException ex) {  
        }  
    }  
}
```

- Il metodo **sleep** è statico e agisce sul thread corrente, anche se lo si invoca su un oggetto Thread, non agirà su di esso!!
  - nessun Thread può addormentarne un altro
- La chiamata a **sleep** può essere posizionata ovunque, perché tutto gira in un thread

# Esempio di join

```
public class ManyNames {  
    public static void main(String [] args) {  
        NameRunnable nr = new NameRunnable();  
        Thread one = new Thread(nr);  
        Thread two = new Thread(nr);  
        Thread three = new Thread(nr);  
  
        one.start();  
        two.start();  
        three.start();  
        ... ..  
        try{  
            two.join();  
            System.out.println("join con il TWO")  
        }catch (InterruptedException e){}  
        }  
    }
```

Il thread corrente, cioè il **main** attende che il thread **two** termini per poter proseguire nella stampa successiva



# Da Running a Runnable

---

Molto raramente può capitare di dover interagire con lo scheduler per la gestione dei thread.

Il metodo

```
public static void yield()
```

cede il passo ad un altro thread, nel senso che suggerisce allo scheduler di operare lo switch context e mandare in esecuzione uno dei Thread con priorità maggiore o uguale di quello corrente

**Si tratta solo di una proposta!**

Infatti a seguito della chiamata il metodo in running potrebbe essere lo stesso che ha fatto `yield`



# Interruzione

- I metodi `suspend()` e `stop()` sono deprecati perché potrebbero interrompere (anche definitivamente) il thread mentre sta eseguendo un'operazione atomica

- Dalla versione 1.1, sono stati introdotti :

```
public void interrupt()
```

```
public static boolean interrupted()
```

che offrono un meccanismo di interruzione meno «aggressivo», infatti il thread da chiudere viene sollecitato ad interrompersi, ma questo avverrà **solo** se (e quando) esso stesso lo consentirà.

## Funzionamento:

- Il thread che vuole interrompere un altro thread invocherà il metodo `interrupt()` sull'oggetto relativo al thread da chiudere.
  - In questo modo setta una flag che indica la necessità di interrompere il thread
- Il thread che prevede la possibilità di venire interrotto, esegue le operazioni atomiche e, solo alla fine, verifica se ha avuto la richiesta di interruzione con la chiamata a `Thread.interrupted()`.
  - Dopo la chiamata, lo stato del thread viene comunque resettato → la flag viene posta a false

**NB:** il metodo `interrupt` NON determina la chiusura del thread ma imposta solo una flag





---

# La sincronizzazione

---



# Esempio: il c/c

(1)

- La classe Account modella un semplice c/c

```
public class Account {  
    private int balance = 1000;  
    public int getBalance() {  
        return balance;  
    }  
    public void withdraw(int amount) {  
        if(balance >= amount){  
            System.out.println(Thread.currentThread().getName() +  
                                " sta per eseguire il prelievo" );  
            balance = balance - amount;  
            System.out.println("prelievo ok per " +  
                                Thread.currentThread().getName() + ", saldo attuale: " + balance );  
        }  
    }  
}
```

- Il prelievo (***withdraw***) si può fare solo se c'è disponibilità → senza mandare il conto "in rosso"!
- I proprietari **Fred** e **Lucy** condividono il conto e vogliono fare prelievi che andranno oltre la somma disponibile → li modelliamo come 2 istanze della classe **AccountMng** (un thread) che condivideranno il conto



# Esempio: il c/c

(2)

- Il metodo di prelievo esegue questi step
  1. controllare il saldo
  2. se ci sono abbastanza soldi, allora eseguire il prelievo
- Se si separano i 2 step non c'è più coerenza!!!
  - Il controllo al punto 1. perde di senso se nel frattempo lo stato del conto è cambiato!
- Implementiamo una versione di `AccountMng` che non considera il problema degli accessi concorrenti al c/c
  - Dopo mostriamo quella corretta!

# Esempio: il c/c

(3)

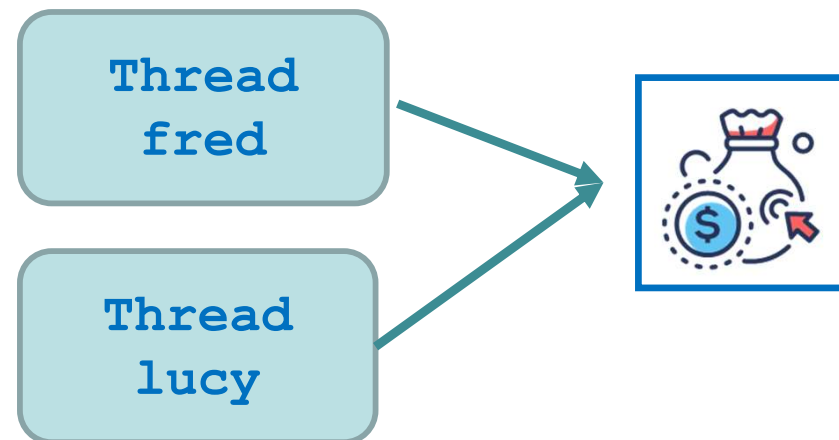
```
public class AccountMng extends Thread{
    private Account cc;
    public AccountMng(String nome, Account cc){
        super(nome);
        this.cc = cc;
    }
    // il metodo run realizza 5 prelievi di 200
    public void run() {
        for (int x = 0; x < 5; x++) {
            cc.withdraw(200);
            System.out.println("saldo attuale: " + cc.getBalance());
        }
    }
}
```

Il gestore del  
conto possiede un  
istanza del conto

# Esempio: il c/c

(4)

```
public class Simulazione {  
    public static void main (String [] args) {  
  
        // creo il conto  
        Account cc = new Account();  
        // creo i 2 thread con la stessa istanza del conto  
        AccountMng fred = new AccountMng("Fred", cc);  
        AccountMng lucy = new AccountMng("Lucy", cc);  
  
        fred.start();  
        lucy.start();  
    }  
}
```





# Esempio: il c/c

(6)

Un possibile output sarebbe:

```
Lucy sta per eseguire il prelievo
Fred sta per eseguire il prelievo
prelievo ok per Lucy, saldo attuale: 800
prelievo ok per Fred, saldo attuale: 600
>>>> Saldo attuale : 600
>>>> Saldo attuale : 600
Fred sta per eseguire il prelievo
Lucy sta per eseguire il prelievo
prelievo ok per Fred, saldo attuale: 400
>>>> Saldo attuale : 200
prelievo ok per Lucy, saldo attuale: 200
>>>> Saldo attuale : 200
```

```
Fred sta per eseguire il prelievo
Lucy sta per eseguire il prelievo
prelievo ok per Lucy, saldo attuale: -200
prelievo ok per Fred, saldo attuale: 0
>>>> Saldo attuale : -200
>>>> Saldo attuale : -200
ammontare non disponibile
ammontare non disponibile
>>>> Saldo attuale : -200
ammontare non disponibile
>>>> Saldo attuale : -200
>>>> Saldo attuale : -200
ammontare non disponibile
>>>> Saldo attuale : -200
```



# Oggetto "occupato"

---


- Lo switch context operato dallo skeduler non tiene conto della logica del metodo prelievo!
- Può accadere infatti che dopo il controllo del saldo, l'esecuzione passi ad un altro thread, mentre controllo del saldo e il prelievo dovrebbero costituire **un'operazione atomica**
- Per ottenere che nessun thread possa invocare il metodo prima che un altro thread abbia finito le operazione sul conto, si può utilizzare la keyword **synchronized**
- In questo modo l'oggetto conto risulta **occupato/bloccato** fino alla fine dell'esecuzione del metodo → l'esecuzione del metodo è asincrona (i thread lavorano uno alla volta!)

# La keyword synchronized

- Un metodo **synchronized**, invocato su un oggetto, può essere eseguito al massimo da un singolo Thread
- Modificando così nell'esempio precedente

```
private synchronized void withdraw(int amount)
```

si ottiene che le operazioni sono consistenti.

- Entrando nel metodo, il thread acquisisce il **lock** sull'oggetto  corrente che ha invocato il metodo → l'oggetto è **bloccato** e nessun altro metodo **synchronized** può essere invocato sullo stesso oggetto.
- Uscendo dal metodo, il thread rilascia il **lock** e i thread che erano in attesa hanno la chance di invocare il metodo (in base all'ordine di arrivo)
- NB: **synchronized** non fa parte della firma, non crea vincoli sugli overriding



# Il lock e synchronized

- Ogni oggetto ha UN solo lock
- Un thread può acquisire **più lock**, MA su **oggetti diversi**
- Un thread in `sleep` non perde il lock sugli oggetti.
- Solo i metodi (e i blocchi di codice) possono essere **synchronized**
  - non le variabili, nè le classi
- Non si sincronizzano i thread, ma gli accessi dei thread sugli oggetti!
- Non è una buona idea impostare tutti i metodi synchronized!
  - La sincronizzazione rende il processo asincrono e, se realizzata male, può causare stati di deadlock!!!




# "A prova di thread"

---

- Una classe è **Thread safe**, se tutti i metodi che accedono a proprietà modificabili dell'oggetto sono stati implementati come metodi **synchronized**
- Nel package delle collection sono thread-safe:
  - **Vector** (implementazione di **List**)
  - **Hashtable** (implementazione di **Map**)
- La classe **Collections** possiede alcuni metodi per rendere sicure le classi che non risultano thread-safe:
  - **static void synchronizedList(List list)**
  - **static void synchronizedSet(Set set)**

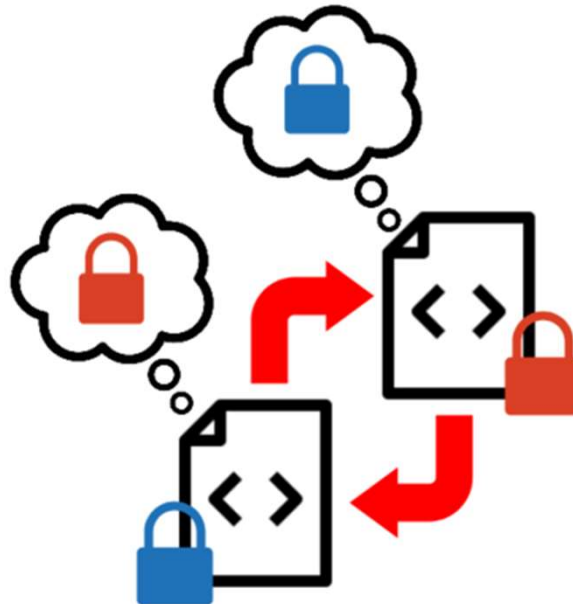
# Se il lock non può essere rilasciato?

- Supponiamo che un thread entra in un metodo sincronizzato e ottiene il lock, ma non ci sono le condizioni per poter eseguire il lavoro.
- Cosa potrebbe fare?
- Opzione 1:
  - esce dal metodo, eventualmente senza lavorare, e rilascia il lock
  - **PROBLEMA**: non sappiamo se esce con lavoro fatto o no!
- Opzione 2:
  - rimane nel metodo, in attesa che qualcosa cambi, NON rilascia il lock
  - **PROBLEMA**: non è detto che la situazione evolva  
→ si potrebbe incorrere nel famigerato **DEADLOCK** 
- **SOLUZIONE**
  - RIMANERE nel metodo, senza lavorare, **E RILASCIARE** il **lock**!
  - I metodi che dobbiamo usare sono **wait()** e **notify()** di **Object**



# DeadLock

- Il Deadlock si verifica quando 2 thread sono bloccati e ciascuno attende il rilascio del lock da parte dell'altro.
- Per evitare il DeadLock bisogna progettare correttamente le azioni dei Thread utilizzando, dove serve, la sincronizzazione





# Waiting e Notifing

- I metodi `wait()` – `notify()` – `notifyAll()` della classe `Object` si possono invocare **solo** all'interno di un contesto `synchronized` → viceversa **`IllegalMonitorStateException`**
- Solo un thread che detiene il lock su un oggetto può invocare `wait/notify` su di esso
  - l'azione però si "scatena" sul thread e non sull'oggetto
- Il metodo `wait()` rende il thread corrente "trasparente" dentro il metodo, non esce, ma rilascia il lock
- Il metodo `notify/notifyAll` invece risveglia uno/tutti thread dal loro stato di waiting
  - il secondo metodo si usa se ci sono diversi thread in attesa
  - se `notify` è invocato in un metodo sincronizzato, il metodo completa il lavoro e poi cede il lock, anche se ha già fatto `notify`!



# Overloading e eccezioni

- Esistono vari overloading di `wait`  
`public final void wait(long timeout)`  
`public final void wait(long timeout, int nanos)`
- che attendono una notifica per il tempo massimo specificato.
- Il metodo `wait` (in tutte le sue versioni) può essere interrotto, proprio come il metodo `sleep` e il metodo `join`
- In tutti questi casi bisogna gestire l'eccezione checked

```
try {  
    wait();           // idem per sleep e join  
} catch (InterruptedException e) {  
    // gestione  
}
```

oppure rilanciarla!

- **NB:** Il metodo `notify` invece non solleva eccezioni checked!