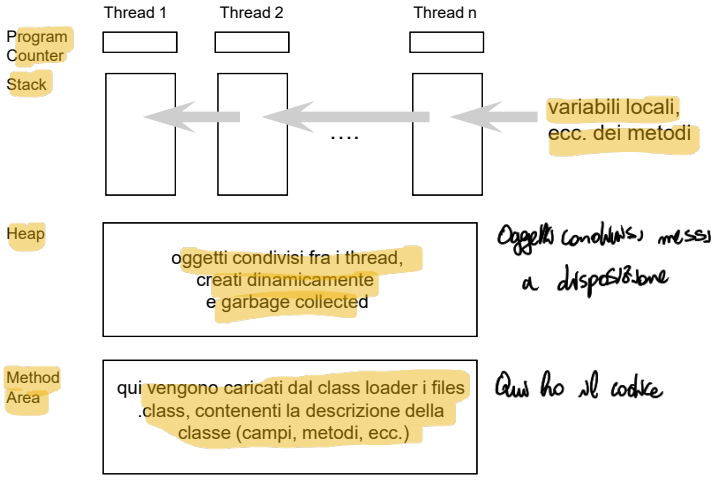
MULTITHREADING

CHE COS'E' IL MULTITHREADING

- possibilità di realizzare più flussi di controllo concorrenti ("threads") nell'ambito di una stessa applicazione
- possibilità di condividere dati fra i threads
- possibilità di controllare i threads e di sincronizzare i threads fra loro, attraverso opportune primitive

RUN-TIME DATA AREA



LA CLASSE Thread

In Java, ogni thread è un oggetto:

- **ogni singolo thread viene creato come una istanza della classe *java.lang.Thread***
- **ogni istanza di *java.lang.Thread* deve essere associata a un metodo *run()* di qualche classe, che contiene il codice che deve essere eseguito dal thread**
- **la classe *java.lang.Thread* contiene tutti i metodi per la gestione dei thread**
- **… e in particolare il metodo *start()*, per "lanciare" il thread con il metodo *run()* "opportuno"**

COME REALIZZARE UN THREAD

Ci sono due metodi per realizzare un thread:

Metodo 1: **estendendo la classe *java.lang.Thread* e sovrascrivendo *run()***

Metodo 2: **implementando la interfaccia *Runnable***

METODO 1

```
class Thread implements Runnable {
        public void start() { …
                run();
        }
        public void run() { }

}
class MyThread extends Thread {
        // eredita start()
        public void run() { < body> }
}

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

sovrascrive

Importante sovrascrivere il metodo run

diventa codice che

Per ogni oggetto ho un thread. Per attivarlo devo chiamare la run.

Sarà eseguito da più Thread. Per attivare più thread devo creare oggetti della classe che ha esteso Thread.

ESEMPIO 1.1

In questo esempio, un main crea e lancia i thread

```
public class drinks {
    public static void main(String[] a) {
            Coffee thc1=new Coffee();  thc1.start();
            Coffee thc2=new Coffee();  thc2.start();
            Tea tht1=new Tea() ; tht1.start();
            Tea tht2=new Tea() ; tht2.start();
    }
}
class Coffee extends Thread {
    public void run() {
            while(true)  { System.out.println("I like coffee");}
    }
}
class Tea extends Thread {
    public void run() {
            while(true)  { System.out.println("I like tea");}
    }
}
```

ESEMPIO 1.2

In questo esempio, ogni thread si lancia da solo

```
public class drinks {
    public static void main(String[] a) {
            Coffee thc=new Coffee();  Tea=tht new Tea() ;
    }
}
class Coffee extends Thread {
    public void run() {
            while(true)  { System.out.println("I like coffee");}
    }
    Coffee() { start(); }
}
class Tea extends Thread {
    public void run() {
            while(true)  { System.out.println("I like tea");}
    }
    Tea() { start(); }
}
```

Nuovo costruttore in cui non si deve usare start

Tutte le variabili nel metodo run sono variabili locali.
Se ho variabile statica, quella è una per tutti i thread.

METODO 2

```
public interface Runnable {
        public abstract void run();
}
class Application implements Runnable {
        public void run() { <body> }
}
...
Application a = new Application();   /* creo l'Application a il cui
                                        metodo run() dovrà essere
                                        eseguita in un thread */
Thread th = new Thread(a);           /* creo un thread e gli passo il
                                        riferimento  all'Application a */
th.start();                          /* lancio il thread, eseguendo il
                                        metodo a.run() */
```

LA CLASSE Thread: SCHEMA

"lancia" il thread, rendendolo schedulabile. Esegue il metodo run() del Thread, che a sua volta lancia il metodo run() dell'oggetto passato come parametro al costruttore Thread(), oppure (se Thread non ha parametri) non fa nulla.

```
public class Thread implements Runnable
        Runnable target;
        ….
        public syncronized native void start() {
        … run();
        }

        Thread (Runnable t) { target = t; …. }

        public void run() { if (target != null) { target.run(); } }
        }

public interface Runnable {
        public abstract void run();
}
```

Uso questo costruttore

Lui genera il valore di target

ESEMPIO 2.1

In questo esempio, un main crea e lancia i thread

```
public class drinks {
    public static void main(String[] a) {
            Coffee  c = new  Coffee();  Tea  t = new Tea();
            Thread th1 = new Thread(c) ; th1.start();
            Thread th2 = new Thread(t) ;  th2.start() ;
    }
}
class Coffee implements Runnable {
    public void run() {
            while(true)  { System.out.println("I like coffee");}
    }
}
class Tea implements Runnable {
    public void run() {
            while(true)  { System.out.println("I like tea");}
    }
}
```

ESEMPIO 2.2

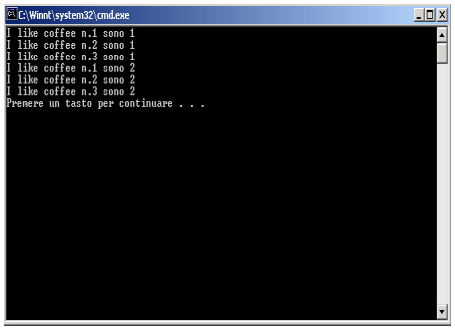
In questo esempio, ogni oggetto crea e lancia il proprio thread

```
public class drinks {
    public static void main(String[] a) {
            Coffee  c = new  Coffee();  Tea  t = new Tea() ;
    }
}
class Coffee implements Runnable {
    Thread th;
    public void run() {
            while(true)  { System.out.println("I like coffee");}
    }
    Coffee() { th = new Thread(this) ; th.start(); }
}
class Tea implements Runnable {
    Thread th;
    public void run() {
            while(true)  { System.out.println("I like tea");}
    }
    Tea() { th = new Thread(this) ; th.start() ; }
}
```

ESEMPIO

```
public class drinks {
    public static void main(String[] a) {
            Coffee  c1 = new  Coffee();
            Coffee  c2 = new  Coffee();
            Thread th1 = new Thread(c1) ; th1.start();
            Thread th2 = new Thread(c2) ;  th2.start() ;
    }
}
class Coffee implements Runnable {
    static int id=0;
    int myid, cont=1;
    public Coffee () {
            id++;
            myid=id;
    }
    public void run() {
     while(cont<=3)  {
        System.out.println("I like coffee n."+ cont+" sono "+ myid);
            cont++;}
    }
}
```

→ è variabile privata



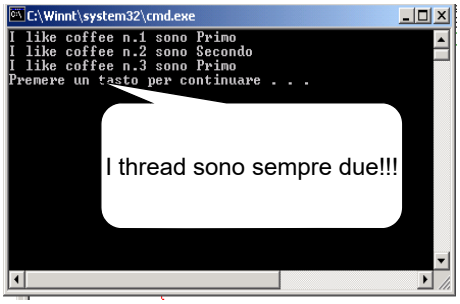
Da evitare: creare un oggetto runnable c1 e poi creare 2 Thread passando stesso oggetto.

Regola: 1 ogg. Thread e 1 runnable

ESEMPIO

```
public class drinks {
    public static void main(String[] a) {
            Coffee  c1 = new  Coffee();
            Thread th1 = new Thread(c1,"Primo") ; th1.start();
            Thread th2 = new Thread(c1,"Secondo") ; th2.start();
    }
}
class Coffee implements Runnable {
    static int id=0;
    int myid, cont=1;
    public Coffee () {
            id++;
            myid=id;
    }
    public void run() {
     while(cont<=3)  {
        Thread t=Thread.currentThread();
         System.out.println("I like coffee n."+ cont+" sono "+ t.getName());
         cont++;
          try {
                    Thread.sleep(100);
                  }
    catch (Exception e){ }
            }
} }
```

↓ Qui do' un nome



Qui le variabili sia statiche che private sono condivise, perché ho passato lo stesso oggetto

Vantaggio del metodo 2

Consente una maggiore flessibilità dal momento che è possibile creare la classe come sottoclasse di qualsiasi altra classe:

```
class EsempioRunnable extends MiaClasse
implements Runnable {
  public void run() {
     .....
  }
}
```

Una classe può estendere solo un'unica classe, quindi se estendo Thread non posso estendere altro. Così posso implementare un'interfaccia Runnable

ESEMPIO

```
class ExThread extends Thread {
   private static int countDown = 5;
   public static void main(String args[]) {
   ExThread t1,t2;
   t1 = new ExThread("Castore");
   t1.start();
   t2 = new ExThread("Polluce");
   t2.start();  }
   public ExThread(String str) {
   super(str); }
   public void run() {
     System.out.println(getName() + ": via!");
     while(true) {
  System.out.println("Thread " +getName() + "(" + countDown + ")");
     if(--countDown == 0) return;
     } }  }
```

Privata, ognuno ha la sua se non ho static

Quando uno dei due non terminerà mai. Accesso all'area condivisa deve diventare protetta. Bisogna che sia eseguito un metodo atomico.

Per fare questo devo usare keyword synchronized. Così accedo a quel posto di codice un thread alla volta.

Bisogna prestare attenzione anche se devo solo leggere

Perché posso decrementare countdown in una parte synchronized ma lo leggo e vengo deschedulato senza decrementare. Dovrei leggere e decrementare.

QUANDO UN thread TERMINA?

Un thread continua nella sua esecuzione, fino a quando:

1. *run()* ritorna
2. viene eseguito il metodo *destroy()* o stop() del thread
3. una eccezione causa l'uscita da *run()*

NOTA: posso passare riferimento a un oggetto nel costruttore di vari oggetti Thread per avere struttura dati condivisa

SCHEDULING

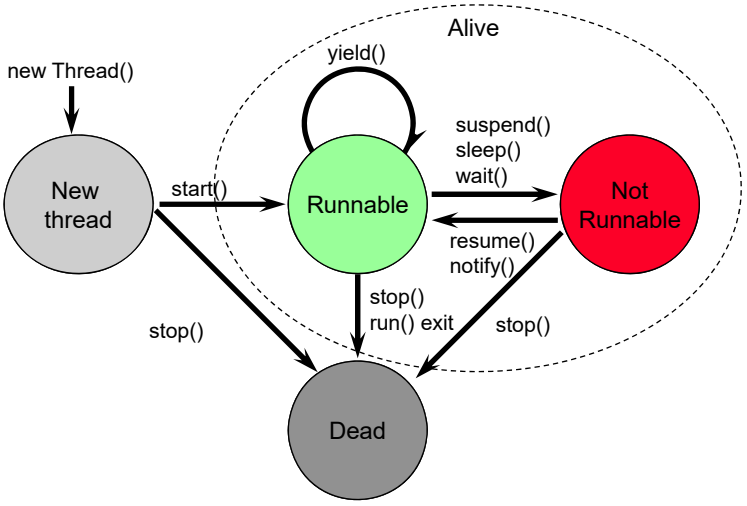
- lo scheduling è effettuato sulla base della priorità dei thread
- la priorità può essere modificata mediante il metodo *setPriority*
- a parità di priorità, lo scheduling è di tipo round-robin

ALCUNI METODI DI Thread

public void **start**()	lancia il thread
public void **run**()	esegue il codice
public final void **destroy**()	distrugge il thread
public final void **stop**() (**D.**)	arresta il thread
public final void **suspend**() (**D.**)	sospende il thread
public final void **resume**() (**D.**)	riattiva il thread
public static void **sleep**(long n)	sospende il thread per n msec
public final void **setPriority**(int newPriority)	modifica la priorità
public final int **getPriority**()	ottieni la priorità
public static void **yield**()	torna in coda di sch.
public final boolean **isAlive**()	true se il thread è vivo

...

GLI STATI DI UN THREAD



METODI synchronized

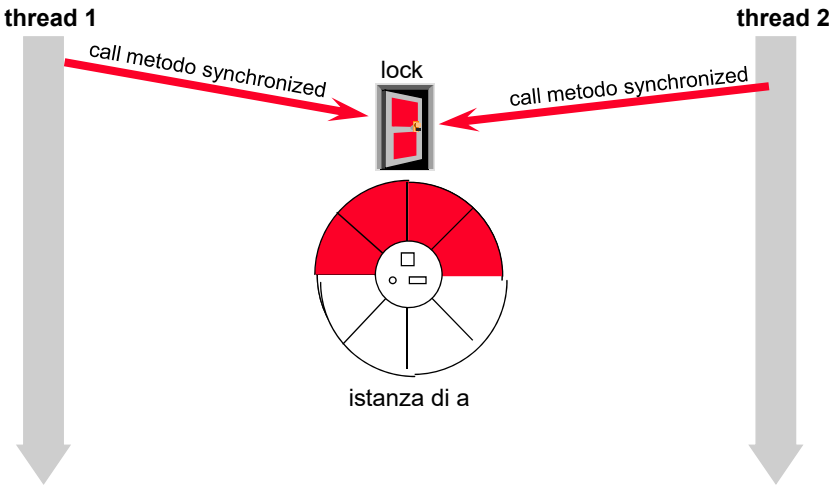
↳ Non sufficiente: è primitivo. Perché? ①

Il modificatore *synchronized* prescrive che il metodo a cui si applica sia eseguito da un solo thread alla volta:

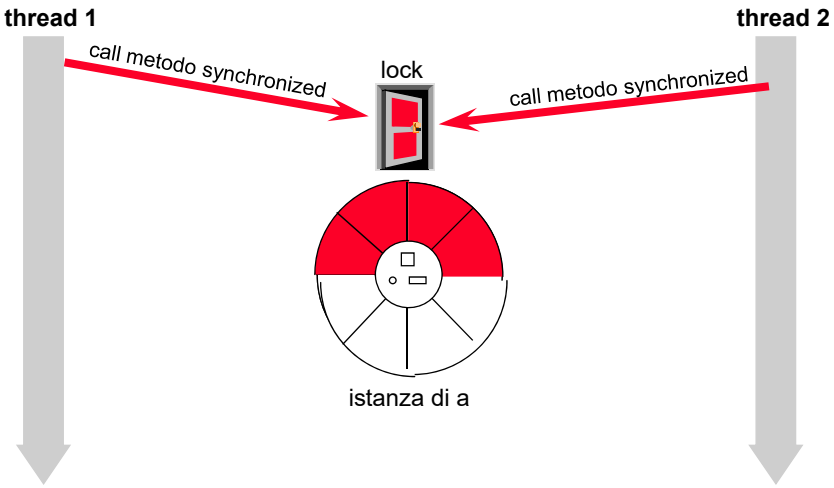
- quando un thread esegue un metodo *synchronized*, viene fatto un lock dell'oggetto cui appartiene il metodo (o, per metodi *static*, della classe)
- quando il thread termina l'esecuzione del metodo, viene fatta una unlock
- prima di tale unlock, eventuali altri metodi synchronized si pongono in attesa

① Immagina una coda con insert protetto da una coda. Operazione sarà fatta una alla volta se anteponge parola chiave synchronized. Java blocca tutto l'oggetto a cui voglio accedere (es. a tutta la coda). Non posso accedere a nessun altro metodo di quell'oggetto. Così restituisce il risultato, ma ha grande effetto collaterale. Metodi synchronized non li usiamo nel modo classico, ma usiamo i semafori. NOTA: Synchronized anche davanti a metodo statico, ma questo blocca tutta la classe, non solo l'oggetto.

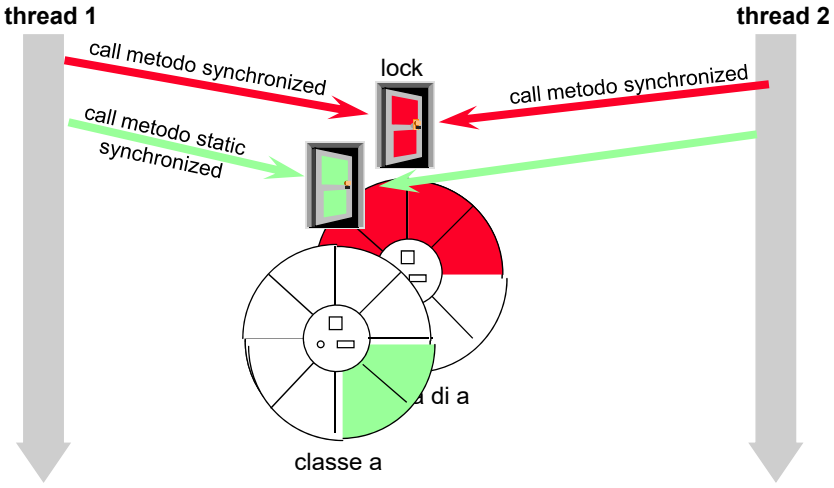
ESEMPIO



ESEMPIO



ESEMPIO



BLOCCHI synchronized

- La keyword *synchronized* può essere usata anche per proteggere *blocchi* arbitrari, e non solo metodi
- in tal caso, occorre specificare l'oggetto di cui acquisire il lock

Esempio:

```
synchronized ( myObject ) { <blocco> }
```

Limita il blocco dell'oggetto sincronizzato.

Thread Main serve a ottenere un oggetto di tipo main, ottenuto con Thread.CurrentThread