**Uso delle classi Timer e TimerTask**

Il timer è un componente hardware del sistema di elaborazione. Contiene alcuni registri utilizzati come *contatori a decremento*, che vengono inizializzati da programma, e possiedono un proprio ingresso di clock, la cui frequenza determina la velocità di decremento del conteggio. Alcuni contatori sono riservati a operazioni di sistema, uno di essi è disponibile per i programmatori. Questo può essere letto durante il conteggio, oppure può essere usato come misuratore di intervallo di tempo. In questo caso, quando il contatore raggiunge il valore 0 il componente avverte la CPU attivando un opportuno segnale di time-out collegato al PIC (controllore delle interruzioni).

La classe seguente, derivata dal gestore del timer, ridefinisce il metodo `run` ereditato. Il metodo `run`, che verrà eseguito quando si verifica il time-out, specifica le operazioni che si dovranno svolgere:

```
import java.util.TimerTask;
  class Scadenza extends TimerTask {
    public void run() {
      System.out.println("Periodo trascorso!");
      this.cancel(); // termina il processo timer
      System.exit(0); // termina anche il processo che lo ha generato.
    }
  }
```

Per essere usata, la classe precedente deve essere ospitata in un processo che, dopo averla *schedulata*, ne consentirà l'esecuzione nel proprio spazio quando se ne presenterà la necessità. La classe seguente `Promemoria` definisce due campi membro:

- `timer` - il riferimento (non ancora inizializzato) a un oggetto di classe timer
- `Avvertimento` - il riferimento a un oggetto di classe `Scadenza`.

Il costruttore della classe inizializza i riferimenti agli oggetti e schedula un processo dopo il numero di secondi indicati nel parametro del costruttore. Il programma principale nel momento in cui crea un'istanza della classe richiama il costruttore.

```
import java.util.Timer;
public class Promemoria {
  Timer timer; // la variabile dovrà contenere un riferimento ad un oggetto di classe Timer
  Scadenza Avvertimento;
  public Promemoria(int secondi) {
    Avvertimento = new Scadenza();
    timer = new Timer(); // la variabile viene inizializzata con il riferimento all'oggetto
    timer.schedule(Avvertimento, secondi*1000);
  }
  public static void main(String args[]) {
  int i=0;
  System.out.println("il processo sta per essere schedulato.");
  new Promemoria(5);
  System.out.println("Processo schedulato. Frattanto faccio altre operazioni");
  do {System.out.print(i+"\r"); if (i<100) i++; else i=0;} while (true);}
}
```

Il metodo `run` comunica il messaggio di promemoria, elimina il processo e termina il programma.

In questo semplice programma sono contenute le parti per implementare e schedulare un processo che deve essere eseguito dal gestore del timer.

- Implementare una sottoclasse della classe `TimerTask`. Il metodo `run` contiene il codice per eseguire il processo. In questo esempio la sottoclasse è chiamata `Scadenza`.
- Creare un thread mediante un'istanza della classe `Timer`.
- Creare un'istanza della classe derivata dal gestore del timer. (`new Scadenza()`).
- Schedulare il processo. L'esempio usa il metodo `schedule`, della classe timer, passando come primo parametro l'oggetto `Avvertimento` di classe `Scadenza` e il ritardo in millisecondi (`5000`), come secondo parametro.

Un altro modo per schedulare un processo è di specificare l'ora in cui si dovrà eseguire un processo. Ad esempio il seguente codice schedula un processo per farlo eseguire alle 23:01:

```
// forma la data corrispondente a 11:01:00 pm di oggi.
Calendar calendario = Calendar.getInstance();
calendario.set(Calendar.HOUR_OF_DAY, 23);
calendario.set(Calendar.MINUTE, 1);
calendario.set(Calendar.SECOND, 0);
Date orario = calendario.getTime();
```

```
timer = new Timer();
timer.schedule(new Scadenza(), orario);
```

Per default, un programma resta in esecuzione fintanto che tutti i suoi processi timer sono in esecuzione. Ci sono quattro modi per terminare un timer thread.

- Richiamare il metodo `cancel` del timer.
- Rendere il thread del timer un "daemon" creando il timer come: `new Timer(true)`. Se i soli thread lasciati dal programma sono daemon thread, il programma termina.
- Dopo che sono terminati tutti i processi del timer schedulati rimuovere tutti i riferimenti all'oggetto `Timer`.
- Richiamare il metodo `System.exit` che fa terminare il programma (e i suoi threads).

L'esempio `Promemoria` richiama il metodo `cancel` dal metodo `run` del processo del timer.

Il metodo `schedule` esiste in varie versioni che, oltre specificare il processo da schedulare, prevedono la possibilità di ripetere la schedulazione:

- `schedule(TimerTask` *`task`*`, long` *`delay`*`, long` *`period`*`)`
- `schedule(TimerTask` *`task`*`, Date` *`time`*`, long` *`period`*`)`
- `scheduleAtFixedRate(TimerTask` *`task`*`, long` *`delay`*`, long` *`period`*`)`
- `scheduleAtFixedRate(TimerTask` *`task`*`, Date` *`firstTime`*`, long` *`period`*`)`

**Personalizzare il metodo `run`.**

La classe `Thread` implementa un processo che non esegue alcuna azione, perché il suo metodo `run` è vuoto. Ci sono due tecniche per fornire un metodo `run` a un thread. Il primo modo per personalizzare un thread è di creare una sottoclasse della classe `Thread` e ridefinire il metodo `run`:

```
public class SempliceThread extends Thread {
  public SempliceThread(String str) {
    super(str);
    }
  public void run() {
    for (int i = 0; i < 10; i++) {
      System.out.println(i + " " + getName());
      try {
        sleep((long)(Math.random() * 1000));
        } catch (InterruptedException e) {}
      }
      System.out.println("FATTO! " + getName());
    }
}
```

Salvare il programma in un file con nome `SempliceThread.java`.

Una delle versioni del costruttore della classe `SempliceThread` riceve una `String` come parametro che viene usata per identificare il thread. In questo esempio imposta il nome del `Thread` che verrà usato più avanti nel programma.

Il metodo successivo nella classe `SempliceThread` è il metodo `run`. Questo è il punto principale di ogni `Thread` e il luogo dove si svolgono le azioni del `Thread`. In questo esempio contiene un ciclo `for` che itera dieci volte. Ad ogni passaggio si mostra il numero dell'iterazione e il nome del `Thread`, quindi trascorre un intervallo di riposo di durata casuale non superiore a 1 secondo. Al termine del ciclo il metodo `run` stampa `FATTO!` Insieme con il nome del thread.

Nel seguente esempio il metodo `main` crea due thread `SempliceThread`: uno è chiamato "Jamaica" e l'altro è chiamato "Fiji". (il programma può essere usato per decidere dove andare in vacanza)

```
public class DueThread {
    public static void main (String[] args) {
        new SempliceThread("Jamaica").start();
        new SempliceThread("Fiji").start();
    }
}
```

Salvare il file con nome `DueThread.java` nella stessa cartella del file precedente. Il metodo `main` avvia i thread chiamando il metodo `start` direttamente dal costruttore. Se il metodo `start` di un thread viene chiamato più di una volta si genera un'eccezione (un errore che può essere gestito da un programma). Notare che l'output di un thread è intercalato con l'output dell'altro thread perché i due processi sono in esecuzione contemporaneamente.

Java usa le eccezioni per fornire la possibilità ai programmi di gestire gli errori che si verificano in fase di esecuzione del programma. Una eccezione è un evento che si verifica durante l'esecuzione di un programma. Le eccezioni devono essere gestite con la coppia `try {…} catch`. All'interno del blocco `try` si scrive un'istruzione Java che potrebbe provocare un errore e, subito dopo, il blocco `catch` specifica un parametro (un gestore dell'errore) e alcune istruzioni che vengono eseguite se si verifica l'errore.

Esercizio: creare un terzo thread.

**Altro esempio di Thread in esecuzione contemporanea.**
Per provare l'esecuzione parallela dei processi l'esempio seguente crea un Thread che genera numeri in un assegnato intervallo, poi avvia due istanze della classe e la stampa mostra che i risultati di un Thread sono intercalati nei risultati dell'altro Thread.

```
public class ThreadContatore extends Thread {
private int inizio;
private int fine;
public ThreadContatore(int da, int a) {
  this.inizio=da;
  this.fine=a;
  }
  public void run(){
  System.out.println(this.getName() + " avviato ... ");
  for (int i=inizio; i<=fine; i++) {
    System.out.print(i + " ");
    }
  System.out.println(this.getName() + " Finito.");
  }
}
```

Per provare il Thread si può usare il seguente programma:

```
public class ThreadTest {
static  public void main(String[] args) {
  ThreadContatore Th1 = new ThreadContatore(1,100);
  ThreadContatore Th2 = new ThreadContatore(200,300);
  Th1.start();
  Th2.start();
  }
}
```

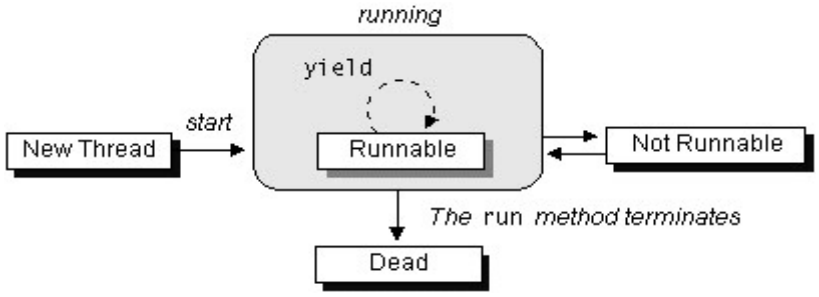
Esercizio. I nomi assegnati per default ai Thread sono `Thread-0` e `Thread-1`. Modificare i nomi dei Thread.

**Implementare l'interfaccia Runnable**
Un'interfaccia definisce un insieme di regole di comportamento che può essere usato da altre classi. L'interfaccia dichiara i metodi (vuoti) e la classe che implementa l'interfaccia deve definirli. Una classe può ereditare metodi e proprietà da una sola classe, ma può avere molte interfacce.

Ci sono due metodi per fornire il metodo run a un thread:

1. Creare una sottoclasse della classe `Thread`, definita nel package `java.lang`, e ridefinire il metodo `run`.
2. Fornire una classe che implementa l'interfaccia `Runnable`, definita nel package `java.lang`, e quindi implementare il metodo `run`, fornito dall'oggetto `Runnable`.

**Il ciclo di vita di un Thread**
La figura seguente mostra gli stati in cui può trovarsi un thread. Sono anche mostrate le cause che provocano i passaggi di stato.




**Creazione del Thread**
Dopo l'esecuzione di `OrologioTh = new Thread(this, "Orologio")`, `OrologioTh` è nello stato New Thread. In questo stato è semplicemente un oggetto `Thread` vuoto; non ha ancora alcuna risorsa. In questo stato si può solo avviare un thread. Qualsiasi altro metodo venga chiamato da questo stato provoca un errore.

**Avvio del Thread**
Adesso si consideri l'istruzione: `OrologioTh.start();` Il metodo `start` crea le risorse necessarie per eseguire il thread, schedula il thread, e chiama il metodo `run` (definito nella classe `Orologio`). Dopo aver eseguito il metodo `start` il thread è in stato "running". Molti computer hanno una sola CPU, quindi i thread non possono essere eseguiti tutti contemporaneamente. Il sistema Java deve implementare uno schema di scheduling che condivide il processore tra tutti i thread "running". In un certo istante un thread è in "running" altri aspettano il proprio turno per entrare in possesso della CPU.

**Rendere un Thread Not Runnable**
un thread diventa Not Runnable quando si verifica uno tra i seguenti eventi:

- Viene richiamato il suo metodo `sleep`.
- Il thread chiama il metodo `wait` per aspettare che si verifichi una certa condizione.
- Il thread è impegnato in operazioni di I/O.

Durante il tempo in cui è sospeso, il thread non è in esecuzione, persino se il processore è disponibile. Al termine dell'intervallo stabilito, il thread diventa Runnable e, se il processore è disponibile, il thread passa in esecuzione.

Per ogni causa che porta nello stato Not Runnable c'è una sola modalità di uscita:

- Se un thread è stato posto in sleep, deve trascorrere il numero di millisecondi specificato.
- Se un thread sta aspettando che si verifichi una certa condizione, allora un altro oggetto deve notificare il cambiamento chiamando `notify` o `notifyAll`.
- Se un thread è bloccato sull'I/O, allora l'operazione di I/O si deve completare.

**Fermare un Thread**
Un thread deve predisporre la propria fine tramite il metodo `run`. La condizione di uscita dal metodo `run` è determinata dalla condizione di chiusura del ciclo `while`.

**Il metodo isAlive**
La classe `Thread` include un metodo chiamato `isAlive`. Questo metodo ritorna `true` se il thread è stato avviato e non fermato. Se il metodo `isAlive` ritorna `false`, il thread o è New Thread o è Terminato. Se il metodo `isAlive` ritorna `true` il thread o è Runnable o è Not Runnable.

**Priorità di un Thread**
In un computer esiste una sola CPU perciò i thread vengono eseguiti uno alla volta ma danno l'illusione della concorrenza. L'esecuzione di più threads su una sola CPU è chiamata *scheduling* che Java realizza tramite un algoritmo deterministico noto come *fixed priority scheduling*.

Quando si crea un thread questo eredita la sua priorità dal thread che lo ha creato. In qualsiasi momento si può modificare la priorità usando il metodo `setPriority`. Le priorità dei Thread sono numeri interi nel range tra `MIN_PRIORITY` e `MAX_PRIORITY` (costanti definite nella classe `Thread`). Quando più threads sono pronti il sistema sceglie quello a più alta priorità. Solo quando il thread si ferma, cede la CPU o diventa not runnable, un thread a priorità più bassa verrà eseguito. Se due thread della stessa priorità sono in attesa della CPU lo scheduler ne sceglie uno con il metodo round-robin. Il thread selezionato resterà in esecuzione fino a quando si verifica una delle seguenti condizioni:

- Un thread a priorità maggiore diventa runnable.
- Esegue `yields` o il suo metodo `run` esegue `exit`.
- Scade il quanto di tempo.

L'algoritmo di scheduling è *preemptive* (a prerilascio). Se, in qualsiasi momento, un thread con priorità maggiore di tutti gli altri threads runnable diventa runnable, il sistema manda in esecuzione il nuovo thread. Si dice che questo thread a priorità maggiore fa *prerilasciare* gli altri thread.

**Sincronizzazione dei Thread**
I thread visti finora non condividevano dati perché contenevano i dati e i metodi richiesti per entrare in esecuzione e non avevano bisogno di risorse o metodi esterni e non erano nemmeno interessati allo stato o all'attività degli altri thread.

Per esempio un'applicazione java ha un thread (il produttore) che scrive dati in un file mentre un secondo thread (il consumatore) legge i dati dallo stesso file. Oppure un thread produttore legge i caratteri dei tasti battuti sulla tastiera e memorizza i codici in una coda e un altro thread consumatore legge i codici dalla coda. Entrambi gli esempi condividono risorse. I thread che condividono risorse devono essere sincronizzati.

**Il problema del Produttore e del Consumatore**
Il produttore genera un intero tra 0 e 9, lo memorizza in un oggetto `Cassetta`, e stampa il numero generato. Per evidenziare il problema della sincronizzazione il Produttore si ferma per un tempo casuale compreso tra 0 e 100 millisecondi prima di ripetere la generazione di un numero:

```
public class Produttore extends Thread {
  private Cassetta cassetta;
  private int numero;
  public Produttore(Cassetta c, int numero) {
    cassetta = c;
    this.numero = numero;
  }
  public void run() {
    for (int i = 0; i < 10; i++) {
      cassetta.deposita(i);
      System.out.println("Produttore #" + this.numero + " inserisce: " + i);
```

```
      try {
        sleep((int)(Math.random() * 100));
      } catch (InterruptedException e) { }
    }
  }
}
```

Il consumatore legge tutti gli interi da `Cassetta` appena questi sono disponibili.

```
public class Consumatore extends Thread {
  private Cassetta cassetta;
  private int numero;
  public Consumatore(Cassetta c, int numero) {
    cassetta = c;
    this.numero = numero;
  }
  public void run() {
    int valore = 0;
    for (int i = 0; i < 10; i++) {
      valore = cassetta.preleva();
      System.out.println("Consumatore #" + this.numero + " preleva: " + valore);
    }
  }
}
```

In questo esempio il `Produttore` e il `Consumatore` condividono l'oggetto `Cassetta`. Si deve notare che il consumatore non si assicura di prelevare un valore solo una volta. La sincronizzazione tra questi due thread avviene a basso livello, nei metodi `preleva` e `deposita` dell'oggetto `Cassetta`. Comunque si assuma per adesso che questi due threads non si preoccupino di sincronizzarsi e si esaminino le possibili conseguenze.

Se il produttore è più veloce del consumatore e genera due numeri prima che il consumatore ne legga uno allora il primo viene perso. Mentre se il consumatore è più veloce del produttore legge più volte lo stesso valore.

Si vuole che il consumatore legga tutti i messaggi prodotti esattamente una sola volta. I problemi sono prevenuti sincronizzando la memorizzazione di un dato nell'oggetto `Cassetta` da parte del produttore con l'operazione di lettura da `Cassetta` da parte del consumatore. Ecco la definizione dell'oggetto `Cassetta`:

```
public class Cassetta {
  private int messaggio;
  private boolean disponibile = false;
  public synchronized int preleva() {
    disponibile = false;
    notifyAll();
    return messaggio;
  }
  public synchronized void deposita(int valore) {
    messaggio = valore;
    disponibile = true;
    notifyAll();
  }
}
```

La sincronizzazione avviene in due possibili modi. Primo, i due threads non devono accedere simultaneamente a `Cassetta`. Un thread può impedirlo bloccando un oggetto. Quando un oggetto è bloccato da un thread e un altro thread tenta di chiamare un metodo sincronizzato sullo stesso oggetto, il secondo thread si bloccherà fino a quando l'oggetto diventerà sbloccato.

Secondo, i due thread devono coordinarsi. Cioè il produttore deve avere un modo per indicare al consumatore che il valore è pronto e il consumatore deve avere un modo per indicare al produttore che il valore è stato prelevato. La classe `Thread` fornisce alcuni metodi -- `wait`, `notify`, e `notifyAll` -- per consentire ai thread di aspettare il verificarsi di una condizione e notificare agli altri thread il momento in cui la condizione si è verificata.

**Il Programma principale.**

Ecco un esempio di problema del produttore e del consumatore che si scambiano messaggi tramite un oggetto comune. L'output di questo programma mostra l'alternanza tra la produzione e la lettura di un dato.

```
public class PrCTest {
  public static void main(String[] args) {
    Cassetta c = new Cassetta();
    Produttore p1 = new Produttore(c, 1);
    Consumatore c1 = new Consumatore(c, 1);
    p1.start();
```

```
    c1.start();
    }
}
```

**Bloccare un oggetto.**

Le parti di programma che accedono allo stesso oggetto da thread concorrenti separati sono chiamate *sezioni* critiche. Una sezione critica può essere un blocco o un metodo ed è identificata con la parola `synchronized`.

Nell'esempio del produttore e del consumatore i metodi `deposita` e `preleva` di `Cassetta` sono sezioni critiche. Il consumatore non deve accedere a `Cassetta` quando il produttore lo sta modificando, e il produttore non deve modificarlo quando il consumatore sta leggendo il valore. Quindi `deposita` e `preleva` nella classe `Cassetta` devono essere marcati con `synchronized`.

Notare che entrambe le dichiarazioni dei metodi `deposita` e `preleva` contengono la parola `synchronized`. Quindi il sistema associa un unico lock con ogni istanza di `Cassetta` (incluso quella condivisa dal produttore e dal consumatore). Quando si entra in un metodo sincronizzato il thread che lo ha chiamato blocca l'oggetto il cui metodo è stato chiamato. Altri thread non possono chiamare un metodo sincronizzato sullo stesso oggetto fino a quando l'oggetto non viene sbloccato. Così quando il produttore chiama il metodo `deposita`, blocca l'oggetto `Cassetta`, impedendo al consumatore di chiamare il metodo `preleva`:

```
public synchronized void deposita(int valore) {
    // Cassetta bloccato dal Produttore
    ..
}
    // Cassetta sbloccato dal Produttore
```

Quando il metodo `deposita` ritorna, il produttore sblocca `Cassetta`. Analogamente quando il consumatore chiama il metodo `preleva` blocca `Cassetta`, impedendo al produttore di chiamare `deposita`:

```
public synchronized int preleva() {
    // Cassetta bloccato dal Consumatore
    ...
}
    // Cassetta sbloccato dal Consumatore
```

L'acquisizione e il rilascio di un lock viene fatto automaticamente e in maniera indivisibile da Java.

**Uso dei metodi notifyAll e wait**

`Cassetta` memorizza i suoi valori nella variabile membro privata chiamata `messaggio`. `Cassetta` ha un'altra variabile membro privata, `disponibile`, di tipo `boolean`. `disponibile` è `true` quando il valore è stato appena inserito ma non ancora prelevato ed è `false` quando il valore è stato prelevato ma non ancora inserito. Ecco possibili implementazioni dei metodi `deposita` e `preleva` (non funzionanti):

```
public synchronized int preleva() {
  if (disponibile == true) {
    disponibile = false;
    return messaggio;
  }
}
```

```
public synchronized void deposita(int valore) {
  if (disponibile == false) {
    disponibile = true;
    messaggio = valore;
  }
}
```

Nella forma in cui sono implementati i due metodi non funzionano. Osservare il metodo `preleva`. Cosa succede se il produttore non ha inserito niente nell'oggetto `Cassetta` e `disponibile` non è `true`? `preleva` non fa niente. Analogamente se il produttore chiama `deposita` prima che il consumatore legga il valore, `deposita` non fa niente.

Si vuole che il consumatore aspetti fino a quando il produttore metta qualcosa in `Cassetta` e il produttore deve notificare al consumatore quando è avvenuto. Similmente, il produttore deve aspettare che il consumatore prenda un valore (e notifichi al produttore dell'avvenuta lettura) prima di sostituirlo con un nuovo valore. I due thread devono coordinarsi e per farlo possono usare i metodi dell'oggetto `wait` e `notifyAll`.

Ecco le nuove implementazioni dei metodi `preleva` e `deposita`:

```
public class Cassetta {
  public synchronized int preleva() {
    while (disponibile == false) {
      try {
        wait(); // attesa che il produttore inserisca un valore
      } catch (InterruptedException e) {
    }
  }
  disponibile = false;
  notifyAll();// notifica al Produttore che il valore è stato prelevato
  return messaggio;
}
```

```
  public synchronized void deposita(int valore) {
    while (disponibile == true) {
      try {
        wait(); // attesa che il consumatore legga un valore
      } catch (InterruptedException e) {
    }
  }
  messaggio = valore;
  disponibile = true;
  notifyAll(); // notifica al Consumatore che il valore è stato inserito
}
```

Il codice nel metodo `preleva` cicla fino a quando il produttore `fornisce un nuovo valore`. Ad ogni ciclo `preleva` chiama `wait`. Il metodo `wait` abbandona il lock mantenuto dal consumatore su `Cassetta` (permettendo perciò al produttore di bloccare e aggiornare `Cassetta`) e allora aspetta una notifica dal produttore. Quando il produttore mette qualcosa in `Cassetta`, lo notifica al consumatore chiamando `notifyAll`. Il consumatore allora esce dallo stato di attesa, `disponibile` è `true`, il ciclo termina, e il metodo `preleva` restituisce il valore contenuto in `Cassetta`.

Il metodo `deposita` funziona in modo simile, aspettando che il thread consumatore legga il valore corrente prima di consentire al produttore di produre un nuovo valore.

Il metodo `notifyAll` visita tutti i thread in attesa sull'oggetto in questione (in questo caso `Cassetta`). Un thread è risvegliato e gli altri restano in attesa. La classe `Object` definisce il metodo di notifica e contiene non solo la versione di `wait` che è usata nell'esempio del problema del produttore e del consumatore che aspetta indefinitamente per la notifica, ma anche altre due versioni del metodo `wait`:

**wait(long *timeout*)** Aspetta la notifica o il `timeout` (misurato in millisecondi).
**wait(long *timeout*, int *nanos*)** Aspetta la notifica o il `timeout` (misurato in millisecondi e nanosecondi).

**Note:** i metodi `wait` per sincronizzare i thread possono anche essere usati al posto di `sleep`. Sia `wait` che `sleep` ritardano per la quantità di tempo specificata ma `wait` può essere risvegliato con una notifica mentre un thread in sleeping non può essere risvegliato.

**Evitare Starvation e Deadlock**

Il problema dei cinque filosofi è usato per illustrare vari problemi collegati ai thread sincronizzati che competono per l'uso di risorse che esistono in quantità limitata:

Cinque filosofi sono seduti intorno a un tavolo. Davanti a ogni filosofo c'è un piatto di riso e tra ogni filosofo c'è uno stecchino. Prima che un filosofo possa mangiare un chicco di riso deve raccogliere due stecchini - uno lo prende a sinistra e l'altro lo prende a destra. I filosofi devono trovare un modo per condividere gli stecchini in modo che ognuno possa mangiare.

Una possibile soluzione: un filosofo raccoglie prima lo stecchino alla sua sinistra appena lo trova. Quando ne entra in possesso tenta di raccogliere quello a destra. Quando lo trova lo preleva e può mangiare un chicco di riso. Poi rimette a posto i due stecchini. Si può verificare che tutti i filosofi abbiano prelevato lo stecchino di sinistra e cadono in stallo.

**Processi Concorrenti.**
Nel seguente esempio si simulano le operazioni in un super mercato:

- due cassiere vendono contemporaneamente N articoli dello stesso prodotto;
- ci si aspetta che la quantità di articoli rimasti in deposito sia diminuita di 2·N unità.

L'operazione di aggiornamento è volutamente ampliata. Un processo:

1) legge, dal database, la quantità di scorta e la trasferisce nella variabile `totale`, che è condivisa tra i processi;
2) sottrae la quantità venduta, `quantita`, ed ottiene la nuova quantità di scorta;
3) salva la nuova quantità di scorta nella variabile condivisa.

L'operazione di aggiornamento della quantità di scorta non deve essere interrotta. Ad esempio, se dopo l'operazione di lettura della variabile arriva l'interruzione Hardware generata dal timer, il processo viene interrotto e messo nella coda dei processi pronti. Il sistema operativo passa l'uso della CPU ad un altro processo, che potrebbe svolgere e completare la stessa operazione.

Quando il sistema operativo preleverà dalla coda il processo interrotto e lo porterà in esecuzione, il processo riprende dal punto in cui era stato sospeso: sottrae la quantità di articoli venduti e salva il risultato, sovrascrivendo il risultato ottenuto dall'altro processo.

Per evitare questo errore, le tre operazioni devono essere eseguite in forma indivisibile, cioè le interruzioni devono essere disabilitate prima che un processo inizi l'operazione di aggiornamento, e devono essere riabilitate solo dopo che è stata completata la terza operazione.

**L'applicazione "gestione magazzino"**
Creare un Progetto `Magazzino` (non togliere la marca di spunta "Create main class").

Clic destro su "`Source Package`" nella casella "`Projects`". Scegliere `New` → `Java Class`.

La classe **Deposito**

```
1   public class Deposito {
2     private int totale;
3     public Deposito(int valoreIniziale) {
4       totale = valoreIniziale;
5     }
6     /* synchronized */ public void togli(int quantita) {
7       int temp = totale;
8       /* Simulazione.interruptHW(); */
9       totale = temp - quantita;
10      System.out.println("Matite in deposito: " + totale);
11    }
  }
```

Linea 2: la proprietà della classe `totale` mantiene la quantità di articoli disponibili in magazzino;

Linee 3÷5: il costruttore assegna il valore iniziale alla proprietà della classe;

Linea 6: il metodo `togli` riceve come parametro il numero di articoli venduti che si devono sottrarre dal numero `totale` di articoli disponibili;

Linea 7: all'interno del metodo `togli` si legge, per ipotesi, dal database, il numero degli articoli disponibili e il valore viene copiato in una variabile temporanea,;

Linea 8: si simula un'interruzione delle operazioni;

Linea 9: si calcola il nuovo `totale` e lo si memorizza.

**Modalità di verifica del programma**:
I segni di commento hanno lo scopo di verificare che, senza interruzioni, l'aggiornamento appare corretto.

Come secondo esperimento si toglierà il commento alla linea 8 per simulare un'interruzione. Uno dei due processi, iniziati contemporaneamente, riceverà l'interruzione per timeout e passa in attesa, l'altro completa l'operazione e si verifica l'errore nell'aggiornamento della variabile.

Per correggere questo caso, si deve togliere il commento alla parola `synchronized`, in questo modo si impedisce ad un processo di richiamare il metodo `togli` se è stato richiamato da un altro processo. In altri termini, il metodo `togli`, realizza l'accesso in mutua esclusione alla variabile condivisa.

Per completare il programma si deve creare un Thread.

Clic destro su "`Source Package`" nella casella "`Projects`". Scegliere `New` → `Java Class`.

**La classe Cassiere**

```
1   class Cassiere extends Thread {
2     private String nome;
3     private Deposito scorta;
4     private static final int NUMOPERAZIONI = 1;
5     private static final int QTAARTICOLI = 5;
6     Cassiere(String nm, Deposito qta) {
7       nome = nm;
8       scorta = qta;
9     }
10    public void run() {
11      System.out.println("Cassiera " + nome + " sta per vendere "
                           + QTAARTICOLI + " matite");
12      scorta.togli(QTAARTICOLI);
13      System.out.println("Cassiera " + nome + " ha venduto " +
QTAARTICOLI + " matite");
14      System.out.println("Cassiera "+nome+" ha venduto un totale
di " + NUMOPERAZIONI * QTAARTICOLI + " matite");
15    }
    }
```

Linea 1: la classe è derivata dalla classe Thread;

Linee 2: il nome del cassiere è una proprietà della classe;

Linea 3: `scorta` è il riferimento al deposito, che contiene la variabile condivisa.

Linee 4 e 5: vengono definite due costanti;

Linee 6÷9: il costruttore inizializza i campi membro della classe;

Linee 10÷15: il metodo `run` verrà richiamato quando il thread verrà avviato.

Linea 12: tra le operazioni di stampa del metodo `run` viene richiamato il metodo `togli` dell'oggetto `scorta`.

**Il metodo `main()`.**

Si crea la classe che usa le classi precedenti.

**La classe Magazzino.**

```
1   public class Magazzino {
2     public static void main(String argv[]) {
3       Deposito dp = new Deposito(10);
4       Cassiere c1 = new Cassiere("Maria", dp);
5       Cassiere c2 = new Cassiere("Wanda", dp);
6       c1.start();
7       c2.start();
8     }
    }
```

Linea 3: si crea un'istanza della classe `Deposito`, inizializzandola con 10 articoli di scorta.

Linee 4 e 5: vengono creati due oggetti di classe `Cassiere`, inizializzati con il nome dell'impiegato e con il riferimento all'oggetto di classe `Deposito`, che quindi è condiviso.

Linee 7 e 8: i thread associati ai due cassieri vengono avviati, cioè viene richiamato il metodo `run`.

**La classe Simulazione**

```
1   class Simulazione {
2     public static void interruptHW() {
3       if (Math.random() < 0.5)
4         try { Thread.sleep(200); }
5         catch(InterruptedException e) { }
6     }
7   }
```

La classe `Simulazione` contiene solo il metodo `interruptHW` che, con probabilità 50%, introduce una pausa di 200 ms nell'esecuzione.

Modalità di verifica del programma:

1. Dopo aver compilato le 4 classi che compongono il programma, si mandi in esecuzione la classe `Magazzino`.
2. Dopo aver osservato che i risultati sono corretti, si tolgano i caratteri di commento intorno all'istruzione di chiamata del metodo `interruptHW` della classe `Simulazione`. Si deve notare che, con probabilità del 50%, l'aggiornamento del numero di articoli di scorta risulta sbagliato.
3. Togliere il commento alla parola `synchronized` ed osservare che i risultati sono corretti.

L'istruzione **`new`** crea il nuovo thread. Il metodo **`start`** dell'oggetto di classe thread avvia l'esecuzione richiamando implicitamente il metodo **`run`** del thread. Al termine del codice del metodo **`run`** il thread termina.

**Risorse condivise**.
L'accesso in scrittura ad una risorsa condivisa deve avvenire in mutua esclusione. Java realizza questo meccanismo mediante un monitor, ovvero una sezione che viene usata nello stesso modo in cui si utilizzano i servizi igienici: si richiede la chiave al proprietario della stanza; si apre la porta con la chiave; si entra nella stanza chiudendo la porta a chiave, dall'interno: in questo modo nessun altro può entrare nella stanza; al termine dell'operazione si riapre la porta, si esce e si richiude a chiave dall'esterno; si riconsegna la chiave al proprietario della stanza; solo a questo punto un'altra persona potrà chiedere al proprietario la chiave della stanza.

In Java, un monitor ha una chiave chiamata **lock**. Quando un thread acquisisce il lock di un certo monitor, si dice che il thread è entrato nel monitor, o che possiede il monitor. Altri thread che tentassero di acquisire il lock vengono messi in attesa che il monitor si liberi. Si dice che questi thread sono in attesa (waitirg) sul monitor.

Java associa sempre un monitor invisibile a qualunque oggetto venga creato, anche se quel monitor non verrà mai utilizzato da alcun programma o thread. Per entrare nel monitor, un thread invoca un qualunque metodo marcato con la parola `synchronized`. Per tutto il tempo nel quale il thread sta eseguendo quel metodo, un altro thread che tenti di chiamare qualunque altro metodo sincronizzato sullo stesso oggetto viene sospeso e dovrà attendere che il primo thread lasci il monitor. II primo thread esce dal monitor quando termina l'esecuzione del metodo sincronizzato in cui era entrato.

Se un thread *thread1* vuole accedere in modo esclusivo a un oggetto *oggettoA,* deve eseguire le seguenti operazioni:

1) il *thread1* chiama un metodo *synchronized* dell'oggettoA;
2) l'esecuzione del metodo inizia, e il *thread1* si trova dentro il monitor invisibile *dell'oggettoA;*
3) se altri thread cercano di chiamare un qualunque metodo *synchronized* sullo stesso *oggettoA,* vengono messi in attesa;
4) il *thread1* completa l'esecuzione del metodo; Un altro thread ha la possibilità di entrare nel monitor.

**Comunicazione tra Processi**
In Java i thread possono comunicare tra loro in modo che uno di essi indichi agli altri quando un certo evento o una certa condizione si è verificata. Nella programmazione tradizionale, il controllo che una certa condizione sia vera o falsa viene effettuato tramite il **polling**: per esempio un programma resta impegnato a controllare ripetutamente il valore di una certa variabile booleana fino a che questa assume il valore *true,* e solo a quel punto prosegue nel proprio lavoro.

Un tipico esempio è rappresentato dalle applicazioni basate sul modello **Produttore Consumatore***,* dove un programma genera dei dati, e un altro legge e utilizza i dati prodotti dal primo: per esempio un thread riceve pacchetti di dati dalla rete e un altro li visualizza all'interno della finestra di un browser.

Con la tecnica del *polling*, il consumatore spenderebbe il proprio tempo con un ciclo *while* di interrogazione di una variabile, che il *produttore* modifica quando mette a disposizione nuovi dati da elaborare. Il produttore, una volta consegnati i dati al consumatore, a sua volta potrebbe essere costretto a rimanere in attesa che il consumatore finisca di elaborare i dati ricevuti. Tutto ciò provoca un inutile spreco di tempo di CPU. Un metodo più efficiente consiste nel permettere a un thread di *segnalare* ad un altro thread il momento in cui un certo evento si è verificato o una certa condizione è diventata vera. In questo modo, il thread in attesa può sospendersi, senza sprecare tempo di CPU.

Java realizza un meccanismo di **inter-process communication** basato su tre metodi: *wait, notify e notifyAll.* Questi metodi possono essere chiamati solo dall'interno di metodi sincronizzati, e possono essere invocati su qualunque oggetto. I tre metodi svolgono le seguenti funzioni:

- **wait** fa uscire il thread dal monitor e lo sospende finché un altro metodo non entri nel monitor e chiami il metodo *notify,*
- **notify** attiva il primo thread che ha eseguito una *wait* sullo stesso oggetto;
- **notifyAll** attiva tutti i thread che hanno eseguito una *wait* sullo stesso oggetto. II thread con la priorità più alta viene mandato in esecuzione per primo.

Molti problemi di programmazione di sistema possono essere affrontati facendo riferimento al modello *Produttore-Consumatore*. Per esempio, un programma legge i dati di un conto corrente dal disco affinché un'applicazione bancaria ne calcoli gli interessi, oppure un thread del sistema operativo rende disponibili aree di memoria dell'elaboratore in modo che altri programmi possano utilizzarli per memorizzare le proprie strutture dati.

**Esercizio.**
*Implementare un modello produttore-consumatore nel quale un produttore genera dei valori numerici che il consumatore riceve e utilizza.*

Per semplicità, in questo esempio si ammette che l'area di scambio sia una variabile in grado di contenere un solo dato, anziché un array o una struttura dinamica: il programma si limita a scambiare un dato per volta, memorizzandolo nella variabile locale `unDato` di un oggetto, istanza della classe chiamata `Vassoio`. Il produttore deposita `unDato` sul `Vassoio` e il consumatore lo preleva. Poi il produttore deposita un nuovo dato, e così via.

Il matodo `main` della classe crea un'istanza di vassoio, una classe che gestisce le operazioni di accesso alla variabile condivisa, e un'istanza di `Produttore` e un'istanza di `Consumatore`:

### La classe principale

```
   class ProdCons {
1    public static void main(String args[]) {
2      Vassoio ilVassoio = new Vassoio();
3      new Produttore(ilVassoio);
4      new Consumatore(ilVassoio);
5    }
6  }
```

La linea numero 3 della classe principale richiama il costruttore della classe `Produttore`. Il costruttore riceve, come parametro, il riferimento al vassoio sul quale il thread deposita i dati destinati al consumatore. È lo stesso costruttore che manda il thread in esecuzione.

Il metodo `run` è un ciclo `for` che per dieci volte mette un dato (un numero intero) sul vassoio.

### La Classe Produttore

```
1   class Produttore implements Runnable {
2     Vassoio v;
3     public Produttore(Vassoio unVassoio) {
4       v = unVassoio;
5       new Thread(this, "Produttore").start();
6     }
7     public void run() {
8       // produce 10 numeri interi in sequenza
9       for(int i= 45; i< 55; i++) {
10        v.metti(i);
11      }
12    }
    }
```

La linea numero 4 della classe principale richiama il costruttore della classe `Consumatore`. Anche questo costruttore riceve dal `main` come parametro lo stesso `Vassoio`.

Il metodo `run` per dieci volte preleva dal vassoio il dato depositato dal *Produttore.*

### La classe Consumatore

```
1   class Consumatore implements Runnable {
2     Vassoio v;
3     public Consumatore(Vassoio unVassoio) {
4       v = unVassoio;
5       new Thread(this, "Consumatore").start();
6     }
7     public void run() {
8       for(int i = 0; i< 10; i++) {
9         v.prendi();
10      }
11    }
12  }
```

La classe `vassoio` non ha il costruttore.

Definisce i due metodi per inserire e per prelevare il dato.

### La classe `Vassoio`.

```
1   class Vassoio {
2     int unDato;
3     synchronized public int prendi() {
4       System.out.println("Preso: " + unDato);
5       return unDato;
6     }
7     synchronized public void metti(int d) {
8       unDato = d;
9       System.out.println("Depositato:" + d);
10    }
11  }
```

Sul vassoio viene scambiato un solo dato numerico; i metodi `metti` e `prendi` sono *synchronized* in quanto accedono in modo concorrente allo stesso vassoio e vanno evitate interferenze tra essi.

Il metodo `metti` ricopia nella variabile locale `unDato` il valore del parametro passato `d`. Il metodo `prendi` restituisce il valore numerico in quel momento presente sul vassoio.

Si osservi che:

- il ciclo `for` del `Consumatore` potrebbe trovare per due o più volte lo stesso valore numerico sul vassoio;
- il ciclo `for` del `Produttore` potrebbe mettere sul vassoio un nuovo valore prima che il `Consumatore` abbia avuto il tempo di leggere il precedente: il `Consumatore` perderebbe in questo modo dei dati.

Nello sviluppo dell'esempio non è stato introdotto alcun controllo sul fatto che ogni singolo dato depositato dal `Produttore` venga sempre prelevato dal `Consumatore` (modello Produttore-Consumatore senza garanzia di ricezione). Si accetta che il *Consumatore* possa perdere qualche dato, o che possa prelevare due volte lo stesso dato dal vassoio.

Questo perché è praticamente impossibile che vi sia una commutazione del contesto da *Produttore a Consumatore* esattamente a ogni esecuzione dei rispettivi cicli `for`.

Questo meccanismo è accettabile in tutti quei casi nei quali non è importante che il *Consumatore* riceva una e una sola volta ogni singolo dato generato dal *Produttore.*

Si provi, nel codice del programma main, ad invertire la sequenza di creazione di Produttore e Consumatore, scrivendo

```
public static void main (String args[]) {
  Vassoio ilVassoio = new Vassoio();
  new Consumatore(ilVassoio);
  new Produttore(ilVassoio);
}
```

Come si comporta il programma? Perché?

**Modello Produttore-Consumatore con ricezione garantita. Approfondimenti**

In generale, è importante che ogni singolo dato generato dal produttore sia sempre ricevuto una e una sola volta dal consumatore. Si immagini per esempio che il produttore sia un programma che segnala i pagamenti delle rate di un mutuo e il secondo sia la banca che riceve tali pagamenti: in questo caso non ci può essere né duplicazione né perdita di dati. I prossimi due esempi trattano queste situazioni di ricezione garantita secondo due meccanismi: *polling e inter-process communication.*

Ricezione garantita con la tecnica del polling.

Per garantire che ogni singolo valore prodotto sia consumato una e una sola volta, si può operare aggiungendo al codice un meccanismo di *polling,* basato su una variabile booleana `datoSulVassoio`:

- il metodo `metti` imposta la variabile a `true` non appena viene depositato un valore;
- il metodo `prendi` la imposta a `false` non appena viene prelevato un valore;
- un ciclo `while` in entrambi i metodi permette loro di sincronizzarsi.

Con questa nuova variabile, che si comporta da semaforo, si risolve anche il caso di vassoio vuoto.

Implementazione della classe

```
    class Vassoio {
1     int unDato;
2     boolean datoSulVassoio = false;
3     int prendi() {
4       while (! datoSulVassoio) // attende che vi sia un dato da prelevare
5       ;                        // il ciclo è vuoto
6       System.out.println("Preso: " + unDato);
7       datoSulVassoio = false;
8       return unDato;
9     }
10    void metti(int d) {
11      while (datoSulVassoio) // attende che il dato precedente sia prelevato
12      ;                      // il ciclo è vuoto
13      unDato = d;
14      datoSulVassoio = true;
15      System.out.println("Depositato: " + d);
16    }
17  }
```

La sincronizzazione tra i due metodi viene ottenuta tramite la variabile booleana `datoSulVassoio`.

Si osservi con attenzione che in questo caso è necessario rimuovere la clausola *synchronized* dai due metodi.

Con la presenza della clausola *synchronized*, il primo ciclo `while` che passa in esecuzione non uscirebbe mai dal monitor, **e** l'altro metodo non avrebbe mai l'opportunità di entrarvi e cambiare il valore della variabile `datoSulVassoio`.

Si crea la situazione di Stallo (*deadlock*).

Quando un programma è costituito da diversi thread concorrenti, che condividono varie risorse, è importante garantire l'equità *(fairness)* tra di essi. Un sistema è equo quando permette a tutti i thread di accedere alle risorse e portare avanti il lavoro. Un sistema equo permette di evitare due problemi noti come *deadlock* (punto morto) e *starvation* (inedia).

Si ha una situazione di **starvation** quando uno o più thread non riescono ad accedere a una risorsa e quindi non possono eseguire il proprio lavoro. Si indica con **deadlock** una situazione di starvation indefinita, quando due o più thread sono in attesa di una condizione che non si verificherà mai. Per esempio, il thread 1 è dentro al monitor dell'oggetto A e deve entrare anche nel monitor dell'oggetto B. Nel frattempo, il thread 2 è nel monitor di B e deve entrare nel monitor di A. In questa situazione, ognuno dei due thread impedisce all'altro di progredire.

Per verificare la situazione di *deadlock* nell'esempio precedente, si provi ad aggiungere la clausola *synchronized* ai metodi `prendi` e `metti` nella classe `Vassoio`.

Il programma, presentato, ha caratteristiche di inefficienza: i due cicli *while* consumano una quantità enorme di tempo macchina senza produrre lavoro utile. Le primitive di *inter-process communication* di Java permettono invece di ottenere lo stesso risultato in modo molto efficiente.

Ricezione garantita via inter-process communication.

Il modo corretto di scrivere un programma in Java, per garantire che ogni valore prodotto sia consumato una e una sola volta, consiste nell'utilizzare le primitive `wait` e `notify` per far comunicare tra loro i due metodi `metti` e `prendi`.

Nella classe `Vassoio`, la variabile booleana `datoSulVassoio` indica ai due metodi `prendi` e `metti` se un dato nuovo è disponibile sul vassoio.

Implementazione della classe

```
1   class Vassoio {
2     int unDato;
3     boolean datoSulVassoio = false;
4     synchronized int prendi() {
5       if (!datoSulVassoio)
6         try {
7           wait(); // sospeso in attesa del nuovo dato da leggere
8         } catch (InterruptedException e) {};
9       System.out.println("\t\t\tPreso: " + unDato);
10      datoSulVassoio = false;
11      notify(); // segnala che il vassoio è libero
12      return unDato;
13    }
14    synchronized void metti(int d) {
15      if (datoSulVassoio)
16        try {
17          wait(); // sospeso in attesa che il vassoio si liberi
18    }
19    catch (InterruptedException e) {};
20    unDato = d;
21    datoSulVassoio = true;
22    System.out.println("Depositato: " + d);
23    notify(); // segnala che il nuovo dato è disponibile
24    }
25  }
```

Se la variabile booleana `datoSulVassoio` ha valore *false,* il metodo `prendi` utilizza la primitiva `wait` per passare nella coda dei processi in attesa fino a quando il *Produttore* non segnala che un dato è pronto da prelevare. Quando questo succede, il metodo `prendi` legge il dato e con la primitiva `notify` indica al *Produttore* che il vassoio è libero.

Se la variabile booleana `datoSulVassoio` ha valore *true*, Il metodo resta in *wait* finché il *Consumatore* non indica che il vassoio è libero. A questo punto, il metodo `metti` deposita sul vassoio il nuovo dato e segnala al *Consumatore* con la primitiva `notify` che il nuovo dato è disponibile.

La clausola `synchronized` garantisce che i due metodi accedano al dato sul vassoio e alla variabile booleana `datoSulVassoio` in modo sincronizzato. Ogni singolo dato viene depositato e prelevato una e una sola volta. Eseguendo il programma con queste modifiche si ottiene uno scambio corretto di messaggi tra i due processi.

**Analisi parallela di grandi quantità di dati**

In molti programmi utilizzati in ambito industriale e scientifico, si pone il problema di analizzare ed elaborare grandi quantità di dati, prelevati da un disco, da un array o da un database di grandi dimensioni. In molti casi l'analisi viene effettuata da molti elaboratori sparsi sul pianeta, poiché la quantità di dati da elaborare è enorme.

In questi casi il lavoro viene suddiviso su più thread indipendenti, ognuno dei quali lavora su un sottoinsieme dei dati a disposizione.

*Costruire il programma che permette a più thread di accedere a un grande deposito di dati, ottenere da questo un intervallo di dati da analizzare, effettuarne l'elaborazione, chiedere un nuovo intervallo di dati, e così via, fino a completare l'analisi di tutti i dati del deposito*

Il deposito di dati contiene due attributi:

- **`dati`** rappresenta l'insieme totale dei dati che il deposito contiene e che i thread devono suddividere tra loro per l'analisi;
- **`contatore`** è un numero che indica quanti dati restano da analizzare: all'inizio coincide con il numero totale dei dati presenti in `dati`.

Il metodo `prendi` indica al thread l'intervallo di dati su cui il thread deve lavorare.

**`class`** `Deposito {`

```
1    int dati[];
2    int contatore = 1000;
3    public Deposito() {
4      dati = new int[1000];
5      for (int i = 0; i < 1000; i++) {
6        dati[i] = i;
7      }
8    }
9    synchronized int prendi() {
10     if (contatore > 0) {
11       contatore -= 100;
12       return contatore;
13     } else {
14       return -1;
15     }
16   }
17 }
```

La classe `Analizzatore` utilizza la variabile `deposito`, che fa riferimento al deposito di dati su cui lavorare. Questa classe implementa il metodo `run` che fa svolgere ai vari thread il proprio lavoro. II costruttore crea un thread e lo attiva con il messaggio `start`.

**`class`** `Analizzatore` **`implements`** `Runnable {`

```
1    Deposito d;
2    Analizzatore(Deposito mioDeposito, String nomeT) {
3      d = mioDeposito;
4      new Thread(this, nomeT).start();
5    }
6    public void run() {
7      int indice = 1000;
8      String nomeT = Thread.currentThread().getName();
9      while (indice >= 0) {
10       indice = d.prendi();
11       if (indice >= 0) {
12         System.out.println(nomeT + ": Analizza dati da " +
                                indice + " a " + (indice+99));
13                 // qui va il codice che elabora i dati ricevuti dal Deposito
14         try {
15           // sleep simula il tempo richiesto per l'analisi ed elaborazione reale dei dati
16           Thread.sleep(500);
17         } catch (Exception e) {  }
18       } else {
19         System.out.println("Analizzatore " + nomeT + " ha terminato.");
20       }
21     }
22   }
23 }
```

Per semplicità, nella classe `Deposito` i dati sono rappresentati come un array di 1000 interi: naturalmente a seconda dell'applicazione specifica potrebbero essere contenuti in un file oppure in un insieme di record più o meno complessi. Nell'esempio i dati sono affidati, a blocchi di 100 per volta, ai vari thread di tipo `Analizzatore`, attivati dal `main`.

Il metodo `prendi`, invocato da un thread, restituisce la posizione del primo dei 100 dati su cui il thread deve lavorare. Avendo un array di 1000 interi (da 0 a 999), il primo thread riceve il valore 900 e lavora sui dati da 900 a 999, il secondo riceve il valore 800 e lavora sui dati da 800 a 899, e così via fino all'esaurimento dei dati da elaborare. A questo punto il metodo `prendi` restituisce il valore -1.

Nella classe `Analizzatore`, il metodo `run` ha un ciclo `while` all'interno del quale viene chiamato il metodo `prendi`, viene effettuata l'elaborazione prevista sui 100 dati così identificati (nel programma l'operazione di analisi dei dati viene simulata da una funzione `sleep`), e poi il ciclo riprende finché il metodo `prendi` restituisce il valore -1.

```
    class Elaborazione {
1     public static void main(String args[]) {
2           Deposito mioDeposito = new Deposito();
3           new Analizzatore(mioDeposito, "\tA1");
4           new Analizzatore(mioDeposito, "\t\tA2");
5           new Analizzatore(mioDeposito, "\t\t\tA3");
6     }
7   }
```

Il programma `main` crea un `Deposito` di dati, e poi crea tre istanze di `Analizzatore`, ognuna delle quali genera un thread che esegue il ciclo `while` realizzato all'interno di `run`. Di seguito viene presentato un esempio di output dell'esecuzione del programma: i tre analizzatori creati indicano i dati su cui di volta in volta stanno lavorando e poi terminano con un messaggio all'utente.

```
A1: Analizza dati da 900 a 999
      A2: Analizza dati da 800 a 899
            A3: Analizza dati da 700.a 799
A1: Analizza dati da 600 a 699
            A3: Analizza dati da 500.a 599
      A2;. Analizza dati da 400 a. 499
A1: Analizza dati da 300 a 399
      A2: Analizza dati da 200 a 299
            A3: Analizza dati da 100 a 199
A1: Analizza dati da 0 a 99.
Analizzatore A3 ha terminato
Analizzatore A2 ha terminato
Analizzatore A1 ha terminato
```

Il programma Java proposto legge dalla classe `Deposito`, tramite thread paralleli, un set di valori, sottomultiplo del numero totale di dati da analizzare, fintanto che i valori da analizzare non si siano esauriti. Ipotizzando che il blocco di dati letto non sia un sottomultiplo del deposito, si gestisca correttamente il prelievo dell'ultimo blocco dati, che conterrà un numero di dati inferiore alla capacità del blocco. Si utilizzino delle costanti per i due parametri (dimensione totale del deposito e dimensione del blocco), in modo che si possano poi facilmente modificare questi due dati e provare il programma con diversi valori.

**Gestione sincronizzata di un conto corrente bancario**

L'esempio seguente rappresenta una situazione in cui l'accesso parallelo a risorse comuni richiede la sincronizzazione reciproca dei thread coinvolti

*Costruire un programma che gestisce un conto corrente bancario, effettuando le operazioni di incremento e decremento del saldo, a fronte dei prelievi e dei versamenti dell'intestatario del conto. Le transazioni dell'intestatario devono essere implementate come thread, e deve essere garantita la sincronizzazione dei vari thread, in modo che il saldo del conto corrente sia sempre corretto.*

La classe `ContoCorrente` ha gli attributi `cognome` e `nome` del correntista, oltre al saldo del relativo conto. I metodi `aggiungi` e `sottrai` sono definiti *synchronized* in modo da non creare sovrapposizioni quando il saldo viene aggiornato in forma concorrente da più thread. Il metodo `stampaSaldo` visualizza la situazione del conto corrente.

Le classi `Prelievo` e `versamento` sono definite come *implements Runnable*, e realizzano i thread che svolgono il lavoro di registrazione di prelievi e versamenti che il correntista può effettuare sul proprio conto. Le variabili `contoCorrente`, `cifra` e `motivo` descrivono i dettagli dell'operazione bancaria relativa; il metodo `run`, per ognuna di esse, esegue il lavoro invocando i metodi `aggiungi` e `sottrai` della classe `ContoCorrente`.

**class** contoCorrente {

```
1     String cognome;
2     String nome;
3     int saldo;
4     public contoCorrente(String cog, String nom) {
5       cognome = cog;
6       nome = nom;
7       saldo = 0;
8     }
9     synchronized void aggiungi(int cifra) {
10      saldo += cifra;
11    }
12    synchronized void sottrai(int cifra) {
13      saldo -= cifra;
14    }
15    synchronized void stampa_saldo() {
16      System.out.println("Saldo di " + cognome + " " + nome + ": " + saldo);
17    }
18  }
```

La classe `Prelievo`.

**class** Prelievo **implements** Runnable {

```
1     contoCorrente conto;
2     int cifra;
3     String motivo;
4       Prelievo(contoCorrente unCC, int c, String m) {
5       conto = unCC;
6       cifra = c;
7       motivo = m;
8       new Thread(this).start();
9     }
10    public void run() {
11      try {
12        Thread.sleep((int)(Math.random()*10000));    // attesa fittizia
13      } catch (Exception e) {};
14      conto.sottrai(cifra);
15      System.out.println("Prelievo del sig. " + conto.cognome +
                           ": Euro " + cifra + " Causale: " + motivo);
16        conto.stampa_saldo();
17    }
    }
```

La classe `Versamento`.

**class** Versamento **implements** Runnable {

```
1     contoCorrente conto;
2     int cifra;
3     String motivo;
4     Versamento(contoCorrente unCC, int c, String m) {
5       conto = unCC;
```

```
6       cifra = c;
7       motivo = m;
8       new Thread(this).start();
9     }
10    public void run() {
11      try {
12        Thread.sleep((int)(Math.random()*10000));     // attesa fittizia
13      } catch (Exception e) {};
14      conto.aggiungi(cifra);
15      System.out.println("Versamento di " + conto.cognome +
                           ": Euro " + cifra + " Causale: " + motivo);
16      conto.stampa_saldo();
17    }
18  }
```

Si noti che il costruttore della classe `Prelievo` attiva il thread con il metodo `start`. Si noti anche che il metodo `run`, prima di chiamare il metodo `sottrai`, attende un tempo di durata casuale (qualche secondo).

```
Thread.sleep((int)(Math.random()*10000));
```

Prima di terminare, il metodo `run` visualizza la descrizione dell'operazione e quindi invoca il metodo `stampaSaldo` del conto corrente. Il comportamento della classe `Versamento` è del tutto analogo.

```
class Banca {
1   public static void main(String args[]) {
2     contoCorrente mioCC = new contoCorrente("Rossi", "Mario");
3       new Versamento(mioCC, 1020, "Stipendio");
4       new Prelievo(mioCC, 250, "Bancomat");
5       new Versamento(mioCC, 120, "Contanti");
6       new Prelievo(mioCC, 87, "Bolletta Gas");
7   }
8 }
```

Il `main` crea un conto corrente intestato al signor *Rossi Mario,* e quindi attiva quattro diverse operazioni bancarie. Si osservi che, poiché ogni thread attende un tempo casuale prima di svolgere effettivamente il proprio compito, la sequenza di operazioni sul conto cambia a ogni esecuzione del programma. Ciò che non cambia è il saldo finale del conto corrente: si provi ad eseguire più volte questo programma, verificando che indipendentemente dall'ordine con cui vengono effettuate le operazioni il saldo finale è sempre corretto e consistente.

Il programma gestisce un solo conto corrente, sul quale effettua prelievi e versamenti definiti dal programmatore nella classe main; senza modificare le classi già esistenti (contoCorrente, Prelievo, Versamento) modificare il programma così che possa gestire più conti correnti. Per semplicità, si definisca solo la creazione di nuovi conti correnti, tralasciando la cancellazione di conti esistenti.

**Allocatore di memoria**

Un sistema operativo deve mettere a disposizione dei programmi le aree di memoria, o di disco, che essi richiedono, utilizzano e poi liberano, restituendole al sistema operativo stesso. Si pensi alle aree di memoria che i programmi possono utilizzare per i propri dati dinamici, oppure per memorizzare temporaneamente pacchetti di dati che arrivano dalla rete. Oppure diversi programmi possono condividere un'area di disco su cui memorizzare temporaneamente dei dati che vengono cancellati dopo essere stati elaborati.

Il programma che realizza funzionalità di gestione di aree di memoria condivise viene generalmente chiamato allocatore di memoria *(memory allocator).*

*Implementare un* allocatore *di memoria* che gestisce *10 blocchi di memoria, messi a disposizione di più thread (utilizzatori) che alternativamente richiedono, utilizzano e restituiscono tali blocchi. Si identifichino i vari thread concorrenti con un numero, e si marchi ogni blocco di memoria con I'identificativo del thread che* sta *utilizzando quel blocco.*

In questo esempio si immagina di gestire un'area di memoria contenente 1000 numeri interi, suddivisi in 10 blocchi da 100.

Per tener traccia dei blocchi liberi e di quelli occupati, si crea un indice rappresentato con un array di dieci elementi, uno per ogni blocco. Il valore di un elemento dell'indice è 0 se il blocco corrispondente è libero.

Se un blocco di memoria è stato invece assegnato a un thread utilizzatore, il corrispondente elemento dell'indice riporta il numero che identifica l'utilizzatore stesso.

Memoria disponibile (di dimensione 1000)

| | | |
|---|---|---|
| 0 | libero | Blocco 0 |
| 100 | Assegnato al thread 4 | Blocco 1 |
| 200 | libero | Blocco 2 |
| 300 | Assegnato al thread 8 | Blocco 3 |
| 400 | Assegnato al thread 1 | Blocco 4 |
| 500 | libero | Blocco 5 |
| 600 | libero | Blocco 6 |
| 700 | Assegnato al thread 7 | Blocco 7 |
| 800 | libero | Blocco 8 |
| 900 | Assegnato al thread 4 | Blocco 9 |

Indice dei blocchi

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 0 | 8 | 1 | 0 | 0 | 7 | 0 | 4 |

L'indice dei blocchi riporta lo stato di ogni blocco di memoria. II valore 0 indica un blocco libero, mentre un numero diverso da 0 indica il thread che ha ricevuto dal *memory allocator* quel blocco di memoria.

Ogni oggetto *Utilizzatore* avvia un thread e utilizza(alloca o dealloca) i blocchi di memoria di cui ha bisogno.

La classe `GestoreMemoria` definisce le variabili locali:

- `mem`, contenente la memoria da gestire (di dimensione 1000 in questo esempio);
- `blocchi`, contenente l'indice che riporta i blocchi liberi e quelli occupati.

I metodi da implementare sono i seguenti:

- `allocaMemoria` viene invocato da un thread che necessita di un blocco di memoria, e assegna il primo blocco libero al thread richiedente. Questo metodo ritorna il numero di blocco libero. Se nessun blocco è libero, questo metodo ritorna il valore -1.
- `liberaMemoria` viene invocato da un thread quando esso non ha più bisogno del blocco di memoria che gli era stato allocato. Questo metodo marca come libero (valore zero nell'indice) il blocco "liberato" dal thread.
- `scriviMemoria` scrive un dato numerico intero in una certa posizione del blocco di memoria allocato al thread chiamante;
- `leggiMemoria` legge il dato numerico intero presente in una certa posizione del blocco di memoria allocato al thread chiamante;
- `stampaMemoria` riporta a video il contenuto dell'indice blocchi.

```
class gestoreMemoria {
1     final int DIMMEM = 1000;   // dimensione della memoria disponibile
2     final int DIMBLOCCO = 100; // dimensione del singolo blocco di memoria
3     final int NUMBLOCCHI = 10; // numero di blocchi di memoria disponibili
4     int mem[];                  // spazio di memoria da gestire
5     int blocchi[];              // indice dei blocchi occupati o liberi
6     public gestoreMemoria() {
7       mem = new int[DIMMEM];
8       blocchi = new int[NUMBLOCCHI];
9       for (int i = 0; i < NUMBLOCCHI; i++) {
10        blocchi[i] = 0;     // inizialmente tutti i blocchi sono liberi
11      }
12    }
13    synchronized int allocaMemoria(int id_utilizzatore) {
14      int bloccoLibero = 0;
15      while (bloccoLibero < NUMBLOCCHI) {
```

```
16        if (blocchi[bloccoLibero] != 0) {
17          bloccoLibero++;
18        } else {
19          break;
20        }
21      }
22      if (bloccoLibero < NUMBLOCCHI) {
23        blocchi[bloccoLibero] = id_utilizzatore;  // il blocco di memoria
24        return bloccoLibero; // viene "marcato" con l'identificatore dell'utilizzatore
25      } else {
26        return -1;
27      }
28    }
29    synchronized void libera_memoria(int bloccoRilasciato) {
31      blocchi[bloccoRilasciato] = 0;
32    }
33    synchronized void scrivi_memoria(int blocco, int posizione, int dato) {
34      mem[blocco*DIMBLOCCO + posizione] = dato;
35    }
36    synchronized int leggi_memoria(int blocco, int posizione) {
37      return(mem[blocco*DIMBLOCCO + posizione]);
38    }
39    synchronized void stampa_memoria() {
40      System.out.println("\t\t\t\tStato della memoria: "  + blocchi[0]
41                  +blocchi[1]+blocchi[2]+blocchi[3]+blocchi[4]+
42                  blocchi[5]+blocchi[6]+blocchi[7]+blocchi[8]+blocchi[9]);
43    }
44  }
```

La classe `Utilizzatore` implementa i thread che richiedono l'allocazione e la deallocazione di blocchi di memoria. Essa definisce due variabili:

- `m`, che fa riferimento al *memory allocator,* istanza di `GestoreMemoria`;
- `idUtilizzatore`, il numero assegnato a ogni istanza di `Utilizzatore` (corrispondente a un thread) per identificarla, e per marcare i blocchi di memoria ad esso assegnati. Per semplicità, in questo esempio si assume di avere un massimo di 9 utilizzatori concorrenti, per rendere più facile la lettura di quanto contenuto nell'indice blocchi del gestore di memoria.

Ogni volta che viene creata un'istanza di `Utilizzatore`, viene creato un nuovo thread che esegue il metodo `run` di questa classe.

```
class Utilizzatore implements Runnable {
```

```
1     gestoreMemoria m;
2     int id_utilizzatore;
3     Utilizzatore(gestoreMemoria miaMemoria, String nomeThread, int id) {
4       m = miaMemoria;
5       id_utilizzatore = id;
6       new Thread(this, nomeThread).start();
7     }
8     public void run() {
9       int blk1, blk2;
10      String nomeThread = Thread.currentThread().getName();
11      for (int i = 0; i < 3; i++) {
12        blk1 = m.allocaMemoria(id_utilizzatore);
13        if (blk1 != -1) {
14          System.out.println(nomeThread + " ha allocato il blocco " + blk1);
15        } else {
16          System.out.println(nomeThread + " non ha trovato blocchi liberi");
17        }
18        m.stampa_memoria();
19        blk2 = m.allocaMemoria(id_utilizzatore);
20        if (blk2 != -1) {
21          System.out.println(nomeThread + " ha allocato il blocco " + blk2);
22        } else {
23          System.out.println(nomeThread + " non ha trovato blocchi liberi");
24        }
25        m.stampa_memoria();
```

```
26          if (blk2 != -1) {
27            m.libera_memoria(blk2);
28            System.out.println(nomeThread + " ha liberato il blocco " + blk2);
29            m.stampa_memoria();
30          }
31          if (blk1 != -1) {
32            m.libera_memoria(blk1);
33            System.out.println(nomeThread + " ha liberato il blocco " + blk1);
34            m.stampa_memoria();
35          }
36        }
37      }
38  }
```

Il codice del metodo `run` della classe `Utilizzatore` genera un'attività di allocazione e deallocazione di memoria da parte del thread in esecuzione. Esso ha un ciclo `for` che per tre volte esegue due allocazioni e due deallocazioni di blocchi di memoria, invocando ogni volta il metodo `stampaMemoria` del gestore di memoria. Si osservi che questo metodo visualizza le 10 cifre che rappresentano l'allocazione attuale dei blocchi. Un blocco libero viene indicato dalla cifra 0 mentre una cifra diversa da 0 indica l'identificativo dell'Utilizzatore che ha preso possesso del blocco di memoria.

Il `main` crea un *memory allocator* con il nome `areaMemoria`, e poi crea 8 utilizzatori: in questo modo viene generato un traffico di allocazioni e deallocazioni, fino a saturare i blocchi disponibili.

**public class** Allocatore {

```
1       public static void main(String args[]) {
2           gestoreMemoria areaMemoria = new gestoreMemoria();
3           new Utilizzatore(areaMemoria, "Utilizzatore 1", 1);
4           new Utilizzatore(areaMemoria, "Utilizzatore 2", 2);
5           new Utilizzatore(areaMemoria, "Utilizzatore 3", 3);
6           new Utilizzatore(areaMemoria, "Utilizzatore 4", 4);
7           new Utilizzatore(areaMemoria, "Utilizzatore 5", 5);
8           new Utilizzatore(areaMemoria, "Utilizzatore 6", 6);
9           new Utilizzatore(areaMemoria, "Utilizzatore 7", 7);
10          new Utilizzatore(areaMemoria, "Utilizzatore 8", 8);
11      }
12  }
```