Multi-threading

- I programmi Java possono eseguire thread multipli in modalità di **concorrenza logica**.
  - Esecuzione di fatto simultanea, dati condivisi.
- Come funziona il multi-threading?
  - Per l'utente, più processi "girano" in parallelo.
  - In realtà, è eseguito solo un processo alla volta.
  - Questo vale per macchine con un processore solo, architetture multi-core possono eventualmente eseguire più processi in parallelo, ma comunque in numero limitato (e non molto grande).

Multi-threading

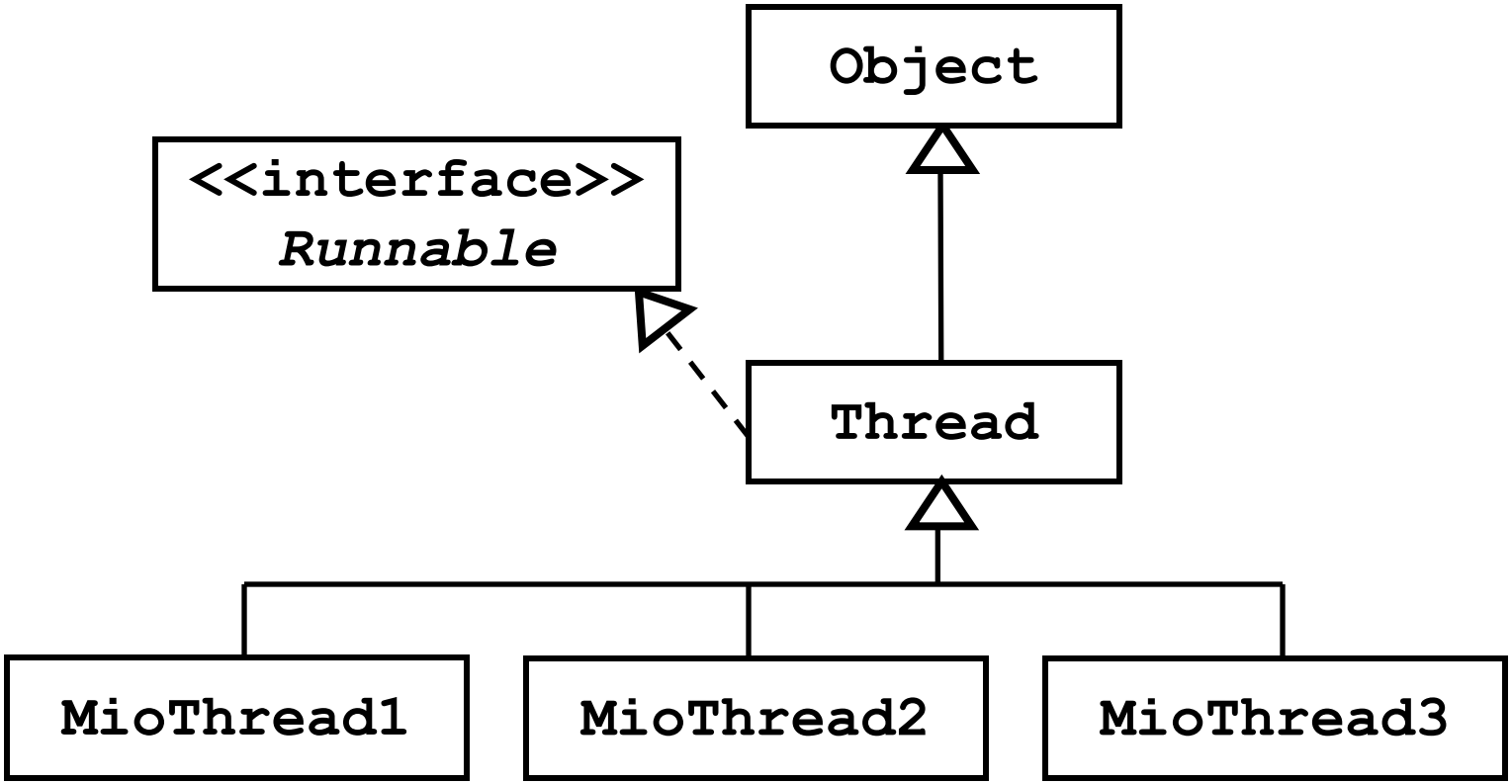
- Per la macchina viene quindi suddiviso il tempo in intervalli piccolissimi detti quanti di tempo, ognuno dedicato a uno specifico processo.
  - Come in un film l'illusione del movimento è data proiettando per breve tempo immagini fisse, la simultaneità segue dalla fine granularità dei quanti.
- La JVM avrà uno scheduler che decide come assegnare ogni quanto di tempo ai thread.
  - I thread alterneranno una fase di esecuzione e una in cui sono pronti ma a riposo (*idle*).

Multi-threading

- Il ciclo standard di vita di un thread prevede perciò: una sua creazione; un'alternanza tra le due fasi esecuzione / pronto; un termine.
  - Sono poi possibili variazioni a questo tipo di ciclo, controllate da appositi comandi.
- In Java è possibile sia creare nuovi thread che fare compiere loro operazioni (alterando il ciclo normale di vita) perché **i thread sono oggetti**.
  - Avremo quindi operazioni che un thread può compiere e messaggi per attivarle: sono definite dalla classe **`Thread`** e dall'interfaccia **`Runnable`**.

Gerarchia dei thread

- La classe **`Thread`** e l'interfaccia **`Runnable`** sono specificate in **`java.lang`**



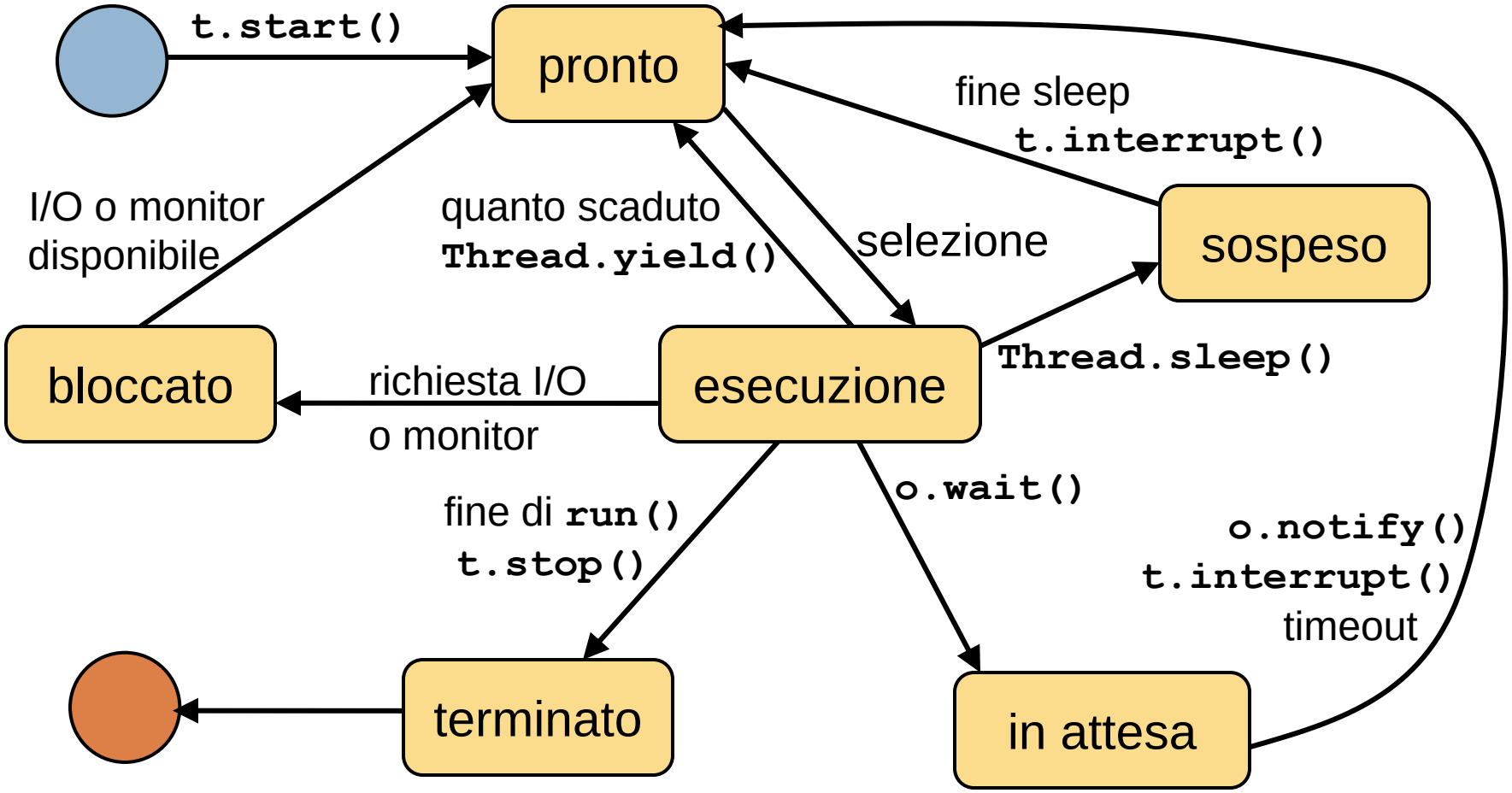
La classe `Thread`

- Possiede i metodi per gestire un thread. In particolare vanno citati:
  - **`run()`**: contiene le istruzioni che il thread esegue; di suo non fa nulla, va sovrascritto nelle estensioni
  - **`start()`**: dà l'avvio al thread
  - **`stop()`**: termina il thread (e fai liberare il suo ambiente dal garbage collector); però è deprecato!
  - **`interrupt()`**: "sveglia" forzatamente il thread
- Questi sono metodi d'istanza, per compiere azioni su un certo thread.

La classe **`Thread`**

- Esistono anche metodi che il thread compie su solo sé stesso. In particolare, sono da invocare come metodi statici di **`Thread`**:
  - **`sleep()`**: sospende il metodo per un ammontare di tempo passato come parametro (in millisecondi)
  - **`yield()`**: esce dall'esecuzione e torna in stato di pronto anche prima della fine del quanto di tempo
  - oltre a vari metodi statici che danno informazioni (tipicamente boolean, oppure **`currentThread`**)
- Invocando questi metodi viene regolato il ciclo di vita di un thread, dall'**inizio** alla **conclusione**.

Il ciclo di vita di un thread



Creare un thread

- All'inizio la JVM crea un solo thread, che esegue il metodo **`main()`** della classe di partenza. Questa è un'istanza della classe **`Thread`**.
  - Per il multi-threading, creiamo altre istanze di **`Thread`**.
- Potremmo invocare il costruttore di **`Thread`** senza parametri (quello di default).
  - Però così il thread avrà il **`run()`** vuoto! Quindi va creata una **sottoclasse** di **`Thread`**: le sue istanze sono anche istanze di **`Thread`**, ma fa override di **`run()`**
  - Non basta: il thread va fatto partire con **`start()`**

Creare un thread

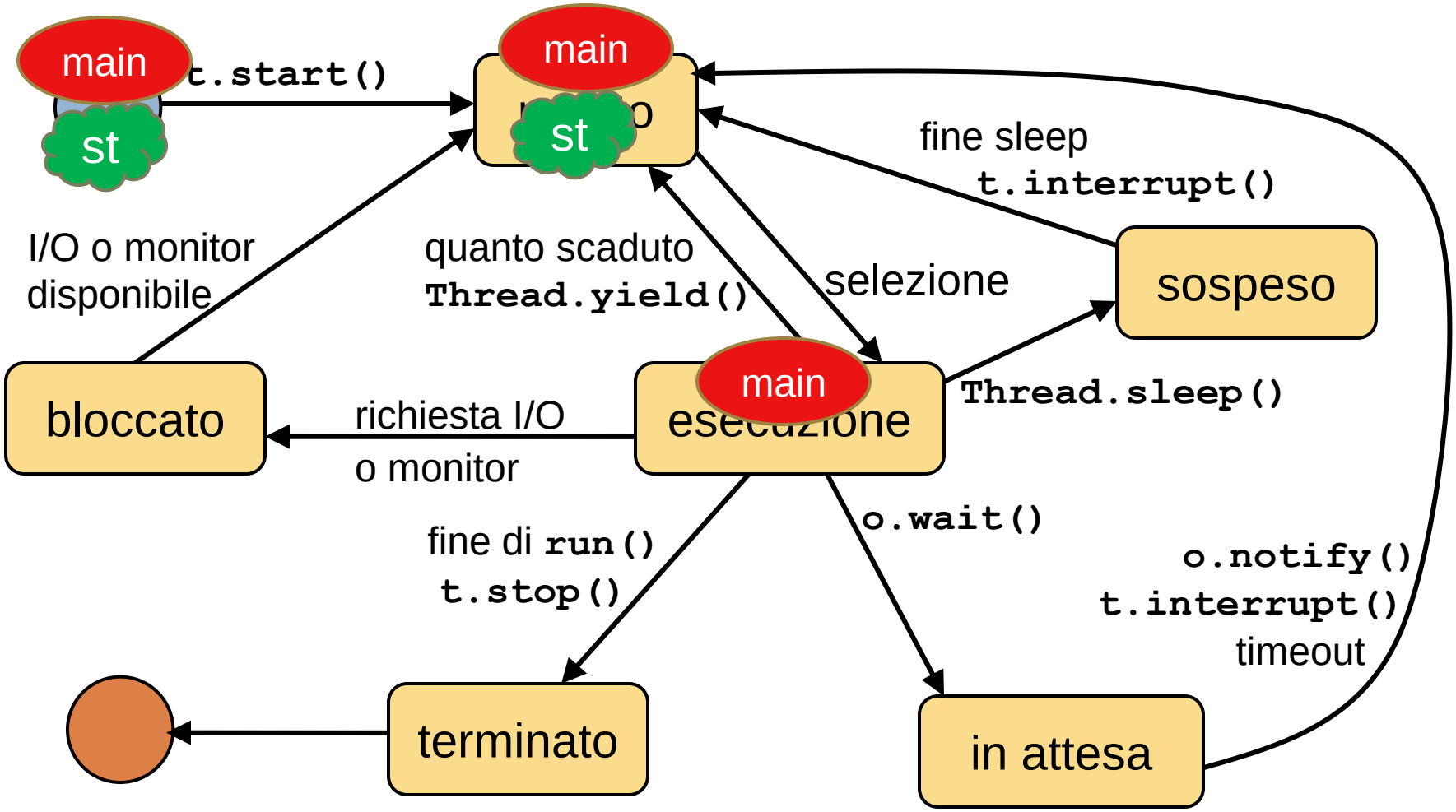
- Un modo di creare un thread allora è di estendere la classe **`Thread`** e sovrascriverne il metodo **`run()`**

```
class Sheepthread extends Thread
{ public void run()
  { int i=2;
    while (!Thread.interrupted() )
    { System.out.println( (i++)+" pecore..”); }
  }
}
public class Contapecore
{ public static void main(String[] args)
  { Sheepthread st = new Sheepthread();
    st.start();
    BufferedReader in = ... ; in.readline();
    st.interrupt();
  }
}
```

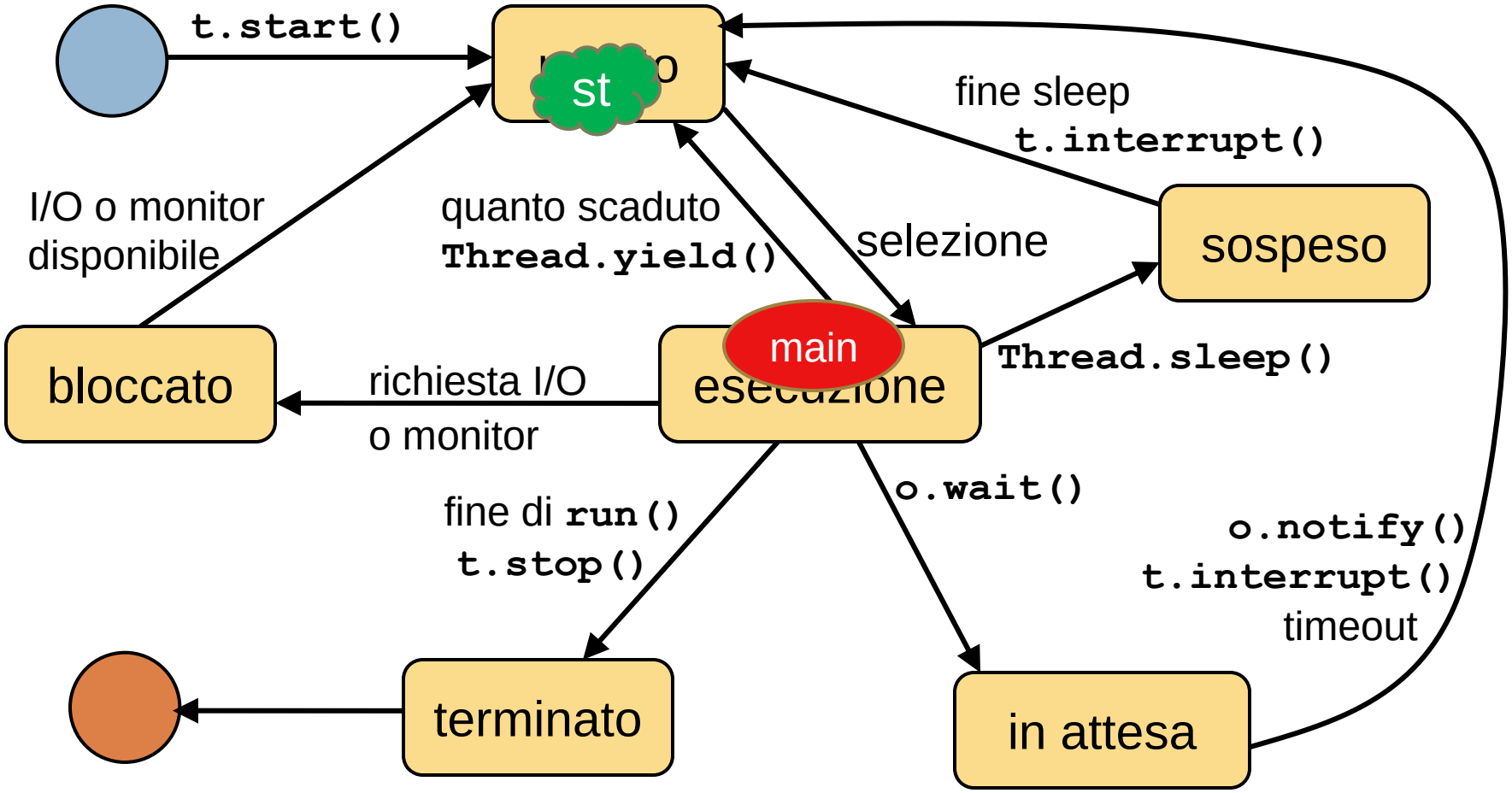
Creare un thread

- Partiamo con un solo thread che esegue **`main`**
- Poi viene creato (e fatto partire) il thread **`st`** di classe **`SheepThread`** (che estende **`Thread`**).
  - Subito dopo `st.start()` abbiamo due thread potenzialmente in esecuzione: ma lo è uno solo (a meno di non avere due processori), l'altro è idle
  - Poi, subito dopo avere attivato **`st`**, il **`main`** richiede un input all'utente (`in.readline()`), quindi diventa **bloccato** e l'unico thread attivo resta **`st`**.
  - Se **`main`** si sblocca esegue `st.interrupt()` che fa terminare **`st`** e poi finisce anche **`main`**.

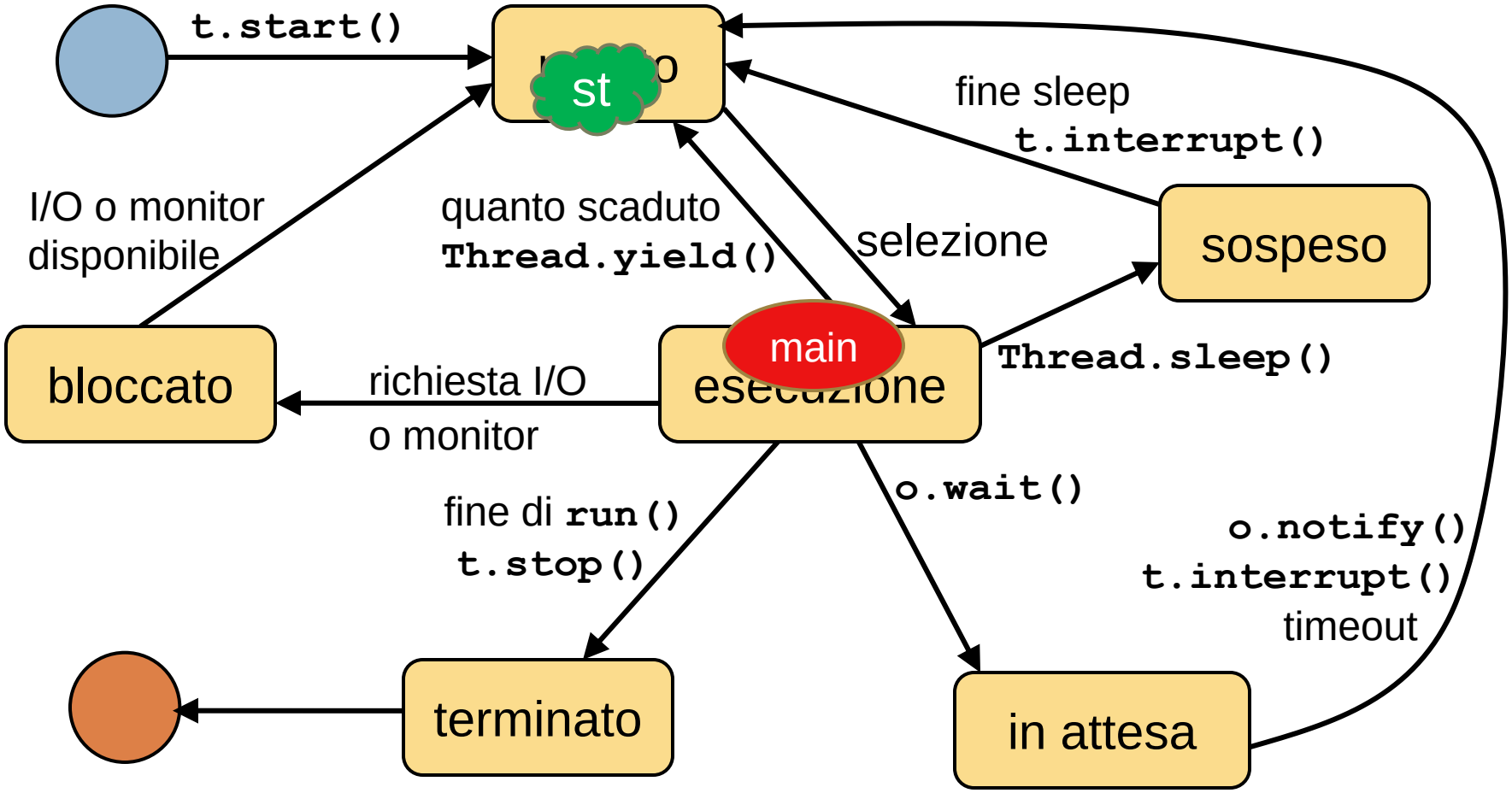
Il ciclo di vita di un thread



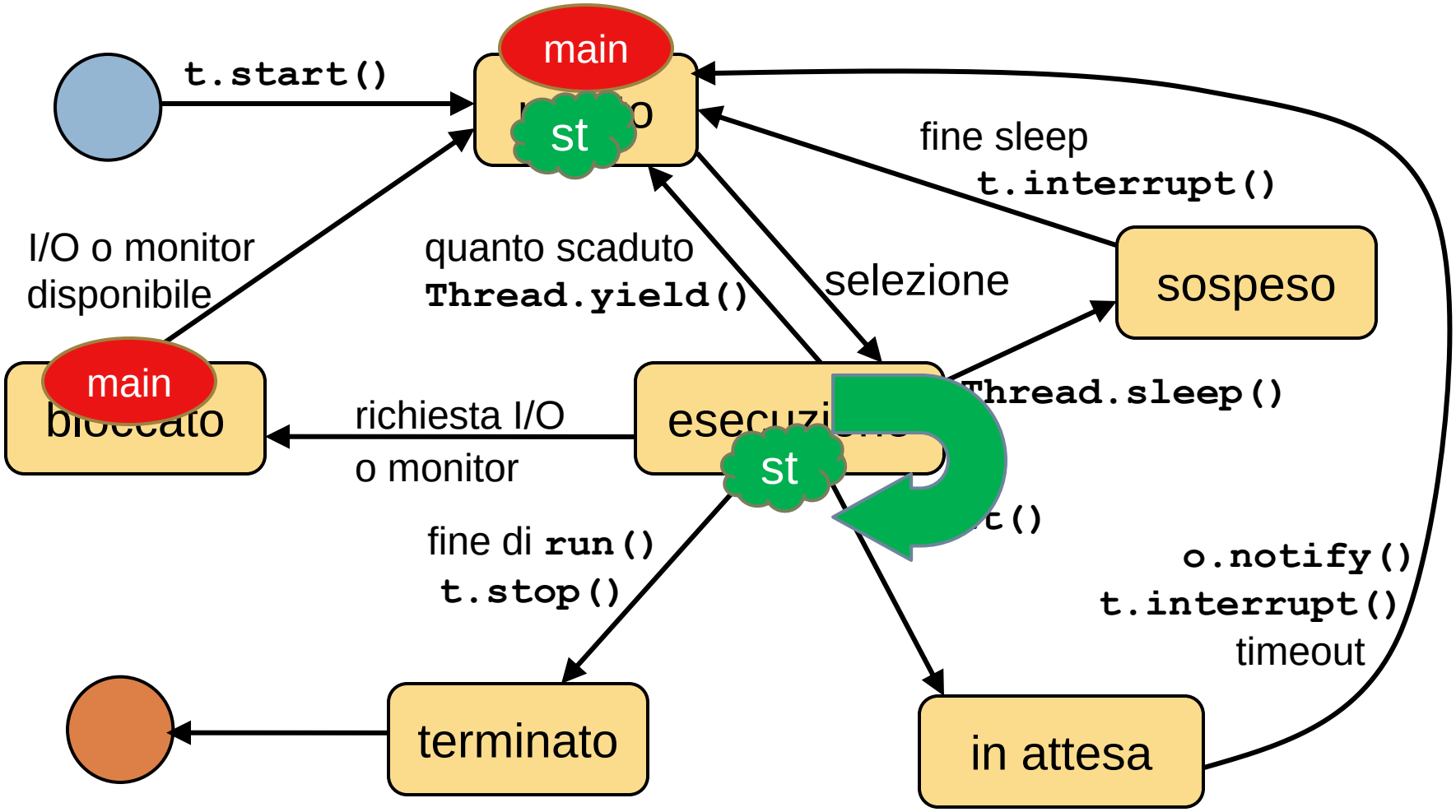
Il ciclo di vita di un thread



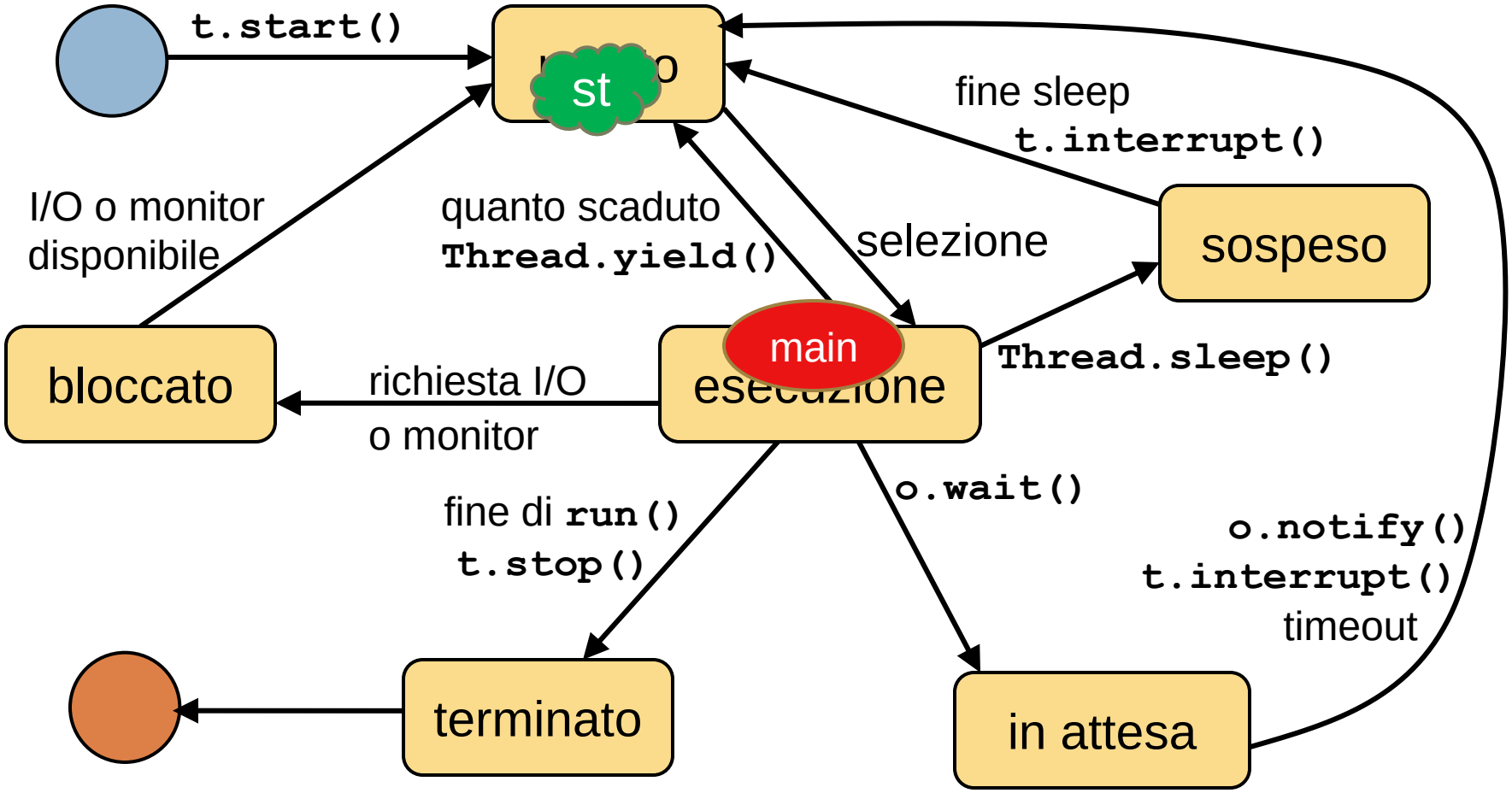
Il ciclo di vita di un thread



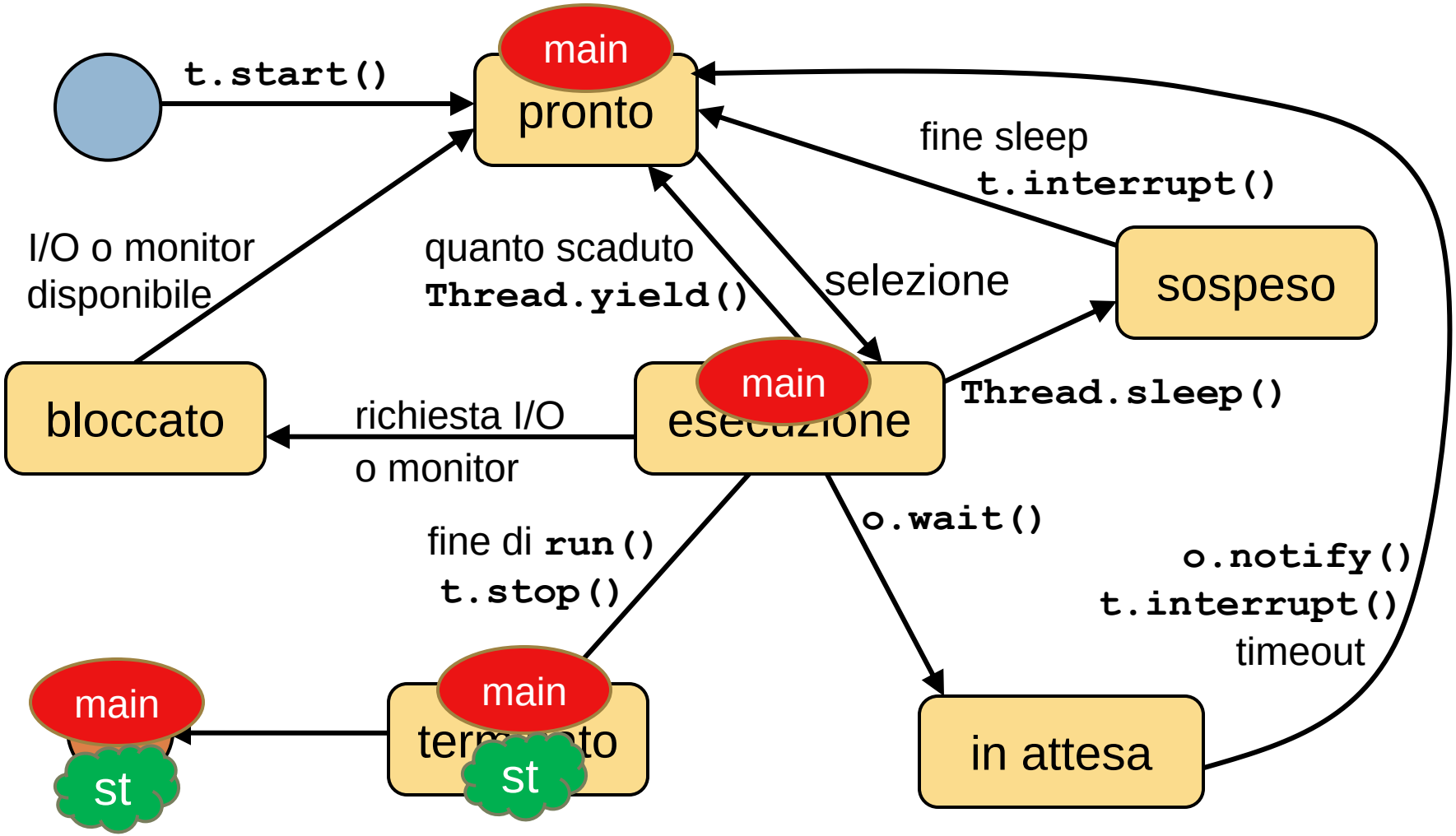
Il ciclo di vita di un thread



Il ciclo di vita di un thread



Il ciclo di vita di un thread



Creare un thread

- In realtà la terminazione forzosa di **`st`** sfrutta un trucco, ossia usa **`st.interrupt`** (corretto semanticamente, ma avrebbe un altro scopo)
  - Si poteva anche usare un altro innesco per questa terminazione, visto che esiste uno spazio dei dati viene **condiviso** dai vari thread: lo heap.
- Da notare che dove viene creato esiste un riferimento al thread "figlio" (la variabile **`st`**)
- Ma se lui vuole un riferimento a sé stesso?
  - Lo ottiene invocando il metodo statico **`Thread.currentThread()`**

Creare un thread

- Questo sistema di creazione thread ha una falla: pensiamo di scrivere un **`-Thread`** estendendo un'altra classe (di cui vogliamo ereditare i metodi).
  - Java non ammette ereditarietà multipla con **`extends`**.
- Tuttavia, viene in aiuto l'interfaccia **`Runnable`**.
- Vale quanto segue:
  - l'unico metodo di **`Runnable`** è **`run()`**
  - la classe **`Thread`** ha anche un costruttore che accetta come parametro un'istanza di **`Runnable`**

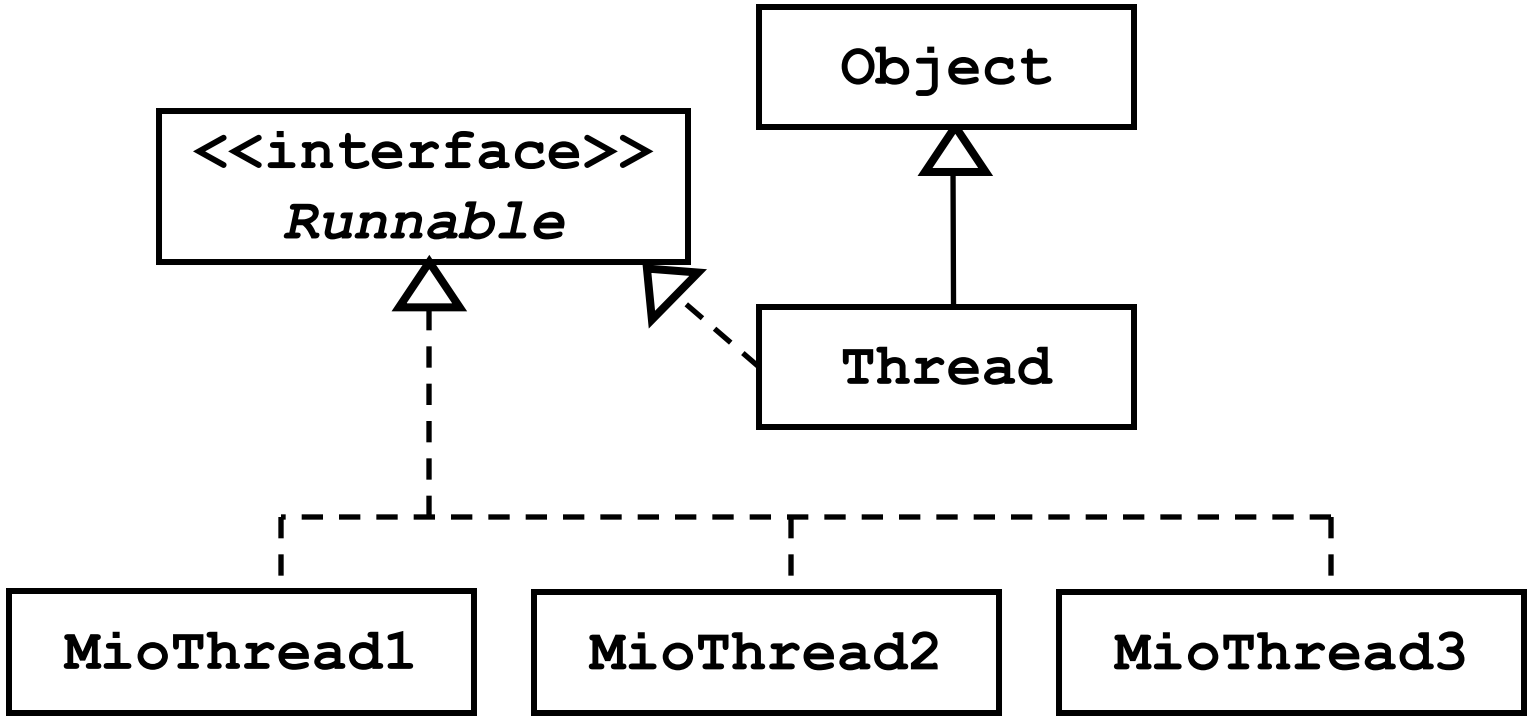
Creare un thread

- Modifichiamo allora l'esempio come segue:

```
class Sheepthread extends Thread implements Runnable
{ public void run()
  { int i=2;
    while (!Thread.interrupted() )
    { System.out.println( (i++)+" pecore..."); }
  }
}
public class Contapecore
{ public static void main(String[] args)
  { Sheepthread st = new Sheepthread();
    Thread th = new Thread(st);
    st.start(); th.start();
    BufferedReader in = ... ; in.readline();
    st.interrupt(); th.interrupt();
  }
}
```

Gerarchia dei thread

- La gerarchia di **`Thread`** e **`Runnable`** cambia.



- Questa modalità è preferibile anche per altri aspetti della programmazione concorrente (che non vedremo).

Thread utente e demoni

- Non è però esatto dire che il thread del **`main`** e gli eventuali altri thread creati dall'utente sono gli unici in esecuzione. Java distingue tra:
  - i **thread utente**, cioè quelli creati con uno scopo specifico dall'utente (il **`main`** ed eventuali altri)
  - i **demoni**, che sono thread di sistema usati dalla JVM per funzioni interne (ad esempio, il garbage collector e il gestore grafico delle finestre)
- I demoni sono invisibili all'utente: ne può essere attivo un numero imprecisato, già fin dall'inizio dell'esecuzione del programma.

Thread utente e demoni

- Non ha senso che un programma esegua solo demoni, visto che hanno funzioni di supporto.
- Un programma “gira” finché ha almeno un thread utente attivo (non per forza il **`main`**).
  - Quando l’ultimo thread utente termina, la JVM distrugge i demoni attivi e chiude il programma.
- Normalmente, le applicazioni generano solo nuovi thread utente. Possiamo però pensare di creare un thread che abbia solo funzioni di servizio, e quindi debba essere un demone.

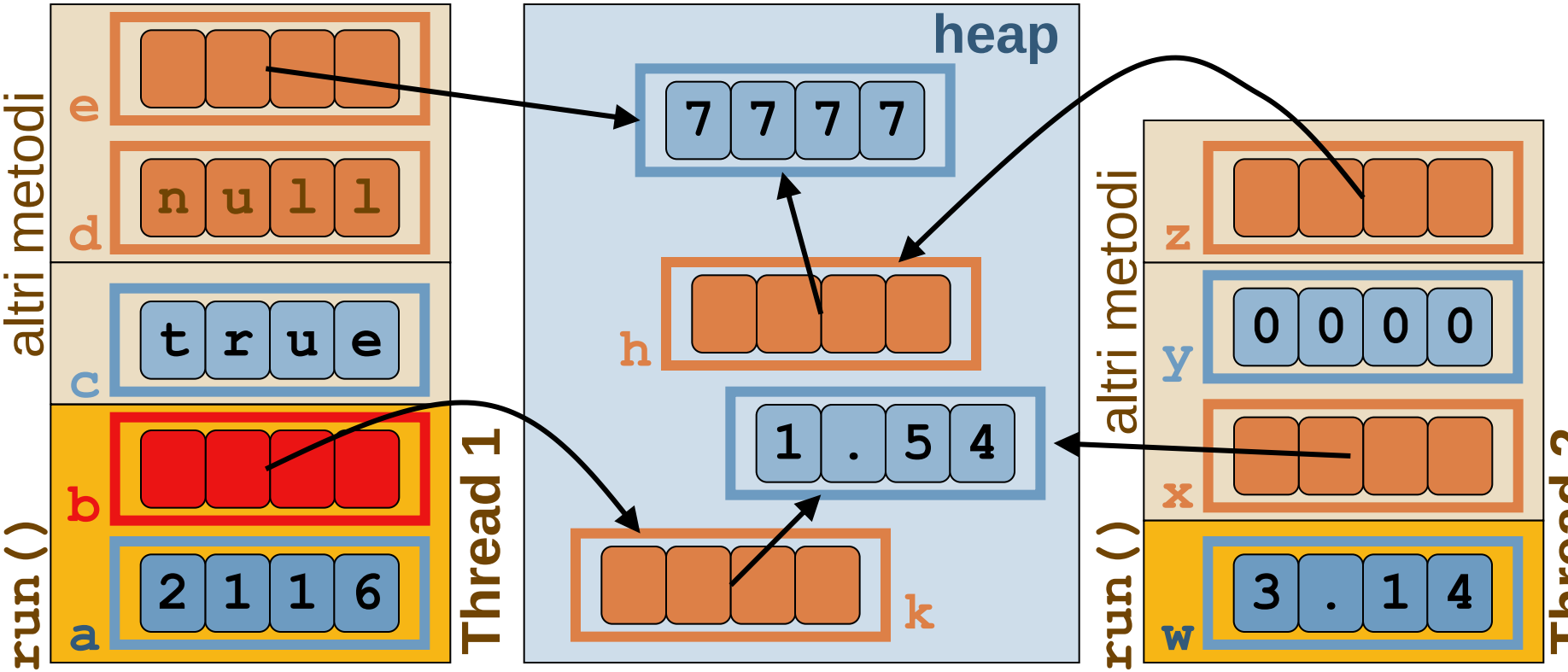
Thread utente e demoni

- Perché definire un thread come demone, anziché come thread utente? Principalmente perché non si vuole che la sua permanenza in attività continui a far eseguire il programma.
- Servono a ciò i metodi d'istanza **`setDaemon()`** e **`isDaemon()`**, che trattano boolean.
- Il "settaggio" a demone va fatto prima di invocare **`start()`**, poi la condizione di demone o thread utente è irreversibile.

Condivisione della memoria

- I vari thread attivi sono tra loro **interdipendenti**.
  - Condividono gran parte della memoria di sistema, in particolare tutto quanto contenuto nello heap, cioè le classi e le loro istanze (oggetti).
  - Invece ogni thread ha una copia privata del suo ambiente (parametri dei metodi e variabili locali).
- Siccome il metodo **`run()`** non ha parametri, l'ambiente di ogni thread è inizialmente vuoto.
  - In realtà il primo thread **`main()`** ha parametri **`args`**
  - Comunque, tipicamente **`run()`** definisce variabili e invoca altri metodi, così l'ambiente si "popola".

Condivisione della memoria



- Ogni thread ha la sua **pila d'ambiente.**
  - Può però accedere tutto quanto viene riferito (anche con più livelli di riferimento)

Condivisione della memoria

- Quindi diversi thread possono avere accesso allo stesso dato. Ciò può creare problemi.
- Pensiamo all'esempio del conto bancario (classe **`BankAccount`**, metodo **`deposit()`**)
  - **`deposit()`** fa **get** del valore di **`balance`**, somma il deposito, fa un **set** al nuovo valore.
  - Supponiamo di eseguire due operazioni di **`deposit()`** tramite thread differenti (per esempio un deposito simultaneo in due filiali). Cosa potrebbe andare storto?

Condivisione della memoria

- Il thread 1 esegue **`deposit(10000)`**, il thread 2 esegue **`deposit(200)`**. **`balance`** vale 500.
- Vengono eseguiti i seguenti passi:
  - Il thread 1 va in esecuzione. Esegue **`getBalance`** e ottiene come valore di ritorno 500. Torna idle.
  - Il thread 2 va in esecuzione. Esegue **`getBalance`** e ottiene come valore di ritorno 500. Torna idle.
  - Ora tocca a 1 che setta **`balance`** a 500+10000
  - Poi tocca a 2 che setta **`balance`** a **500**+200
- Alla fine **`balance`** ha un valore sbagliato!

Condivisione della memoria

- Il problema nasce perché:
  - non è possibile sapere in che ordine i thread eseguiranno le loro operazioni
  - un thread dovrebbe garantirsi diritti esclusivi di modifica: in particolare, in questo esempio, le operazioni di get e set dello stesso thread dovrebbero essere consecutive
- In altre parole il thread dovrebbe riservarsi l'esclusiva per quell'oggetto: solo lui può accederlo (sia per leggerlo che modificarlo)

Monitor

- Questo viene realizzato tramite il meccanismo dei **monitor**. Per un thread, prendere il monitor di un oggetto significa riservarsene l'accesso.
  - è come "piantare la bandierina" o "avere il testimone"
  - Di default, il monitor è libero (nessun thread lo detiene)
- Ogni oggetto possiede uno e un solo monitor.
- Il monitor è una caratteristica definita nella classe `Object`, così ogni oggetto Java ne ha uno, poiché tutte le classi ereditano da `Object`.

Monitor

- I thread possono prendere e rilasciare monitor. Se un monitor è libero, qualunque thread lo può prendere subito. Se è già stato preso, il thread entra nello stato **bloccato** (attende che si liberi)
  - Appena un monitor si libera, tutti i thread che lo vogliono (e che al momento sono bloccati) competono per averlo. Uno solo vincerà, gli altri torneranno bloccati fino alla prossima contesa.
  - Un thread può anche prendere un monitor più volte: avendo lui il monitor ha automaticamente successo a riprenderselo, deve liberarlo più volte perché altri lo possano avere.

Synchronized

- Per cercare di riservarsi il monitor dento a un thread si usa la parola chiave **`synchronized`**.
  Va usata come segue:

```
synchronized (espressione) {blocco}
```

oggetto di cui si vuole il monitor

istruzioni da eseguire una volta che si ha il monitor

- Il monitor viene rilasciato al termine di `blocco`.
  - Dentro a `blocco`, si ha la garanzia che nessun altro thread può usare l'oggetto di cui si detiene il monitor. Chi ci prova entra in **bloccato**.

La classe `Class`

- Le classi (e anche i tipi primitivi, come insieme di dati) possono essere viste **come oggetti**.
- Esiste infatti la classe **`Class`** le cui istanze sono le classi che stanno girando sulla JVM.
  - Non ha costruttori pubblici: non servono, si crea un oggetto istanza appena c'è una nuova classe.
  - Se **`Studente`** è una classe, il suo oggetto classe verrà identificato da **`Studente.class`**
  - È un oggetto e come tale ha un suo monitor

Monitor di classe

- Ci sono monitor anche per le classi oltre che per le loro istanze, per dare l'esclusiva su metodi e campi di classe (cioè statici).
  - Un thread che detiene il monitor per un oggetto istanza ha l'esclusiva per i suoi campi e metodi. Chiunque può invocare gli stessi metodi ma per altre istanze, o i metodi statici di quella classe.
  - Se un thread detiene il monitor per una classe, è l'unico a poterne cambiare i campi statici o invocare i metodi statici (ma tutti possono invocare i metodi di istanza su un qualsiasi oggetto istanza di quella classe).

Monitor

- Per esempio, se vogliamo implementare un sistema di notifiche concorrenti all'utente:
  - abbiamo più thread che stampano a video
  - se non li sincronizziamo, il testo potrebbe risultare ingarbugliato (alternando testi diversi)
- Per evitare questa situazione, non stampiamo a video direttamente, ma mettiamo i messaggi da stampare in coda a una **`LinkedList`**.
  - Ci sarà un altro thread che ogni tanto la svuota.

Monitor

- Sia **`bufText`** il nome di questa lista tampone.
  - È un oggetto, quindi dotato di un proprio monitor.
  - Per “stampare”(ossia inserire in coda a **`bufText`**):

```
synchronized(bufText) { bufText.add(messaggio) }
```

  - Mentre altrove un thread farà eternamente:

```
while (true)
{ synchronized(bufText)
  { if (bufText.size()>0)
    System.out.println(bufText.remove(0));
  }
}
try { Thread.sleep(1000); }
catch (InterruptedException e) { ; }
```

Monitor

- L'ultimo blocco è il **`run()`** di un thread che periodicamente (ogni secondo) controlla se ci sono messaggi e se sì ne stampa uno.
  - Poi aspetta per un secondo in cui non fa altro che catturare (e ignorare) l'eccezione causata da chi gli invoca **`interrupt()`**: così, non finisce mai
  - Ha perciò un ciclo infinito: ma in realtà va prima definito come demone tramite **`setDaemon()`**
  - Così il tutto è corretto, perché la sua presenza non impedisce la terminazione del programma (a quel punto anche lui verrà terminato dalla JVM).

`Synchronized` (ancora)

- Riscriviamo l'esempio di prima.
  - Usiamo una struttura modulare, con una classe **`Notificatore`** che ha opportuni metodi.
  - La classe ha due campi privati, la lista **`bufText`** e un suo thread. Dato che implementa **`Runnable`** potrà costruire i thread che usano il suo **`run()`**.

```
import java.util.*;
public class Notificatore implements Runnable
{ private LinkedList bufText;
  private Thread th;
  // metodi (alla prossima slide)
}
```

Synchronized (ancora)

```
public void addMesg(String s)
{ synchronized(this) { bufText.add(messaggio); } }
public void visualizza()
{ synchronized(this)
  { if (bufText.size()>0)
    System.out.println(bufText.remove(0));
  }
}
public void run()
{ while (true)
  { try { visualizza(); Thread.sleep(1000); }
    catch (InterruptedException e) { ; } }
}
public void avvia()
{ th = new Thread(this);
  th.setDaemon(true); th.start();
}
```

entrambi prendono il monitor dell'istanza di **`Notificatore`** per non essere eseguiti insieme