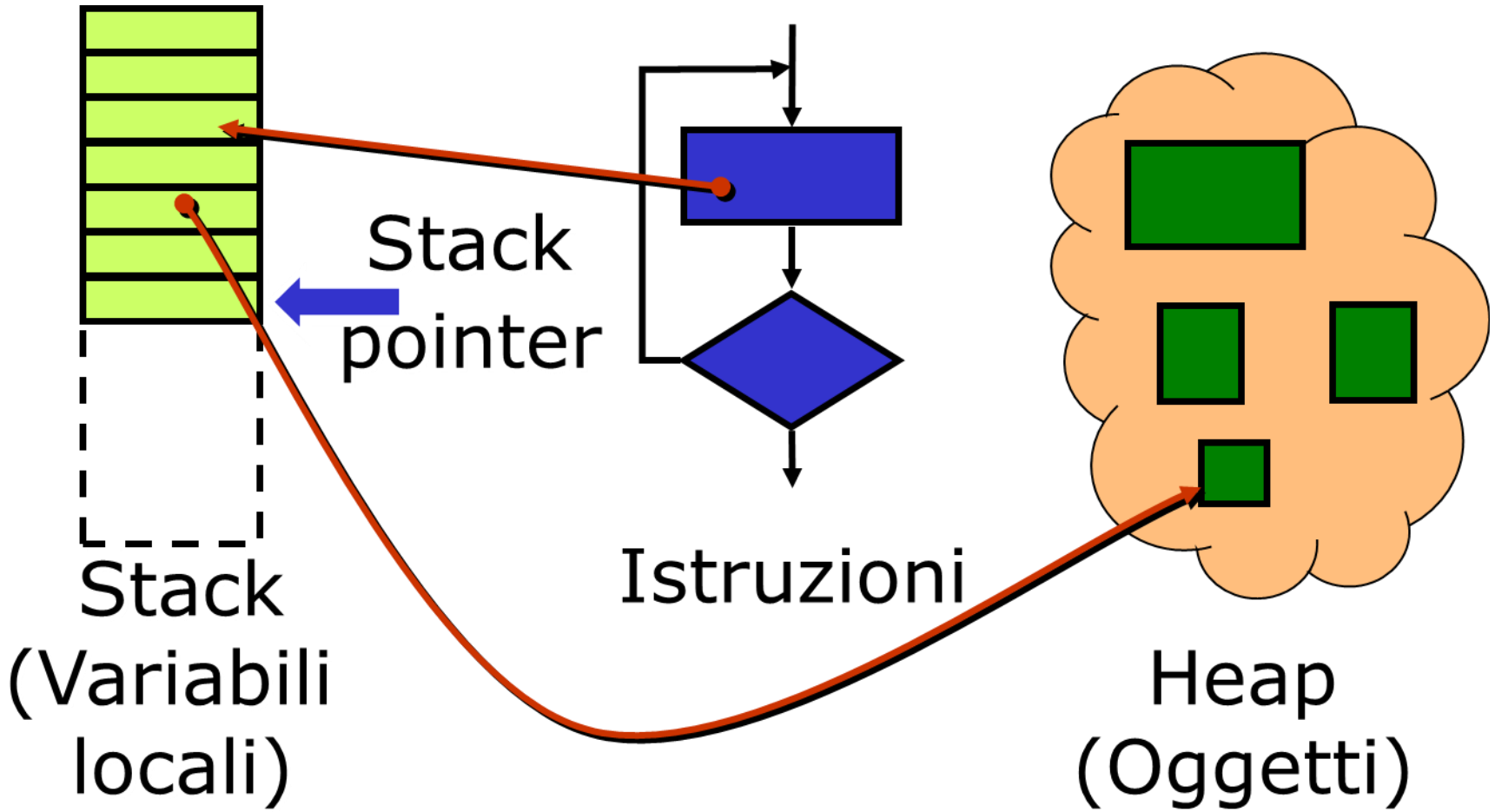
THREAD

- Processi e Thread
- Quando si usano i thread
- Meccanismi di sincronizzazione
- Problemi introdotti dalla programmazione concorrente

Cos'è la programmazione concorrente?

- È un modo di realizzare programmi basato sull'esecuzione – in parallelo – di più **flussi di esecuzione**
  - che co-operano all'interno dello stesso spazio di indirizzamento
- Semplifica la realizzazione di programmi che devono eseguire più operazioni *"allo stesso tempo"*
  - Elaborazione di eventi
  - Richieste di servizio
  - Simulazioni
  - ...

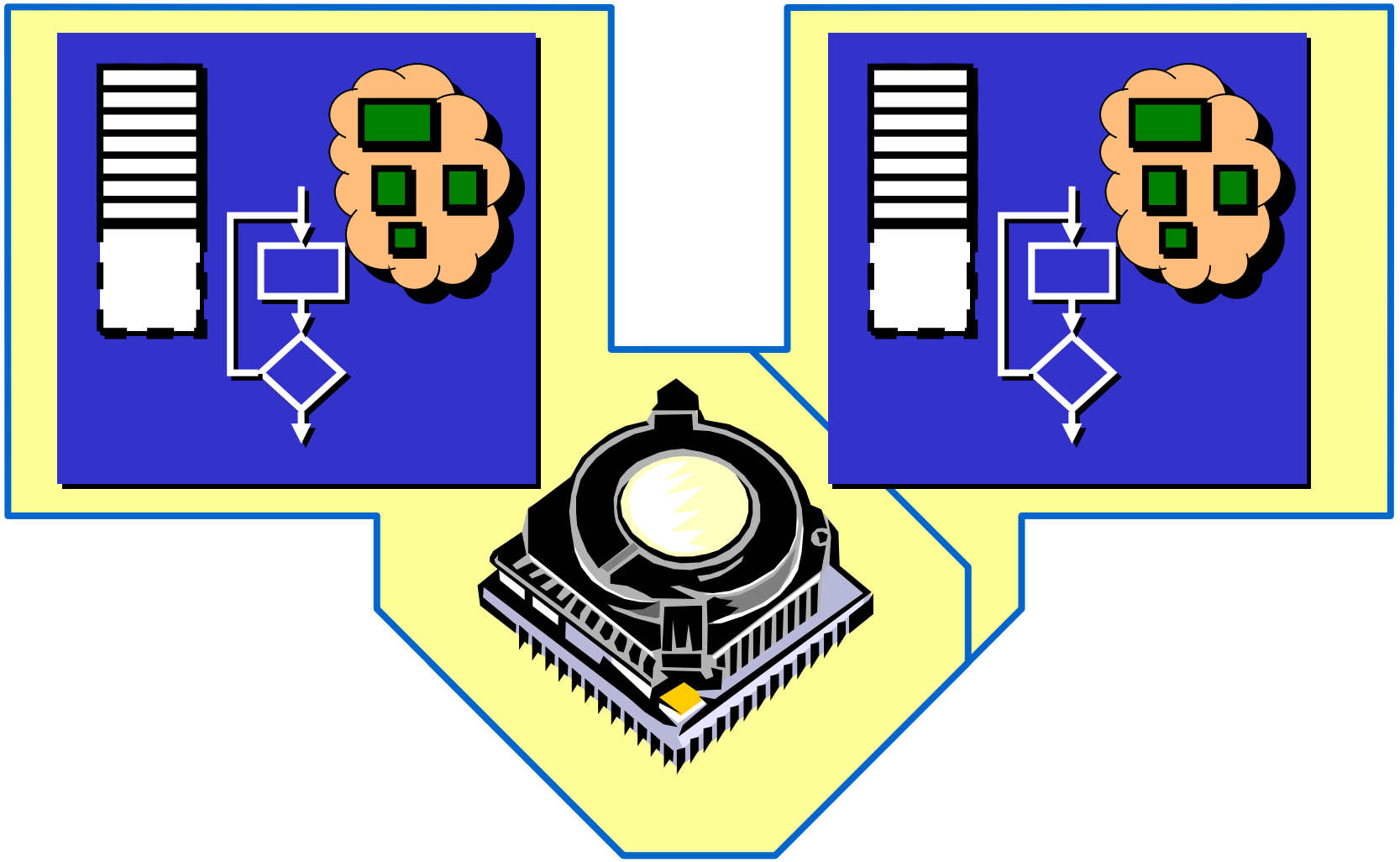
Programmi sequenziali



Simultaneità di esecuzione

- Nei calcolatori dotati di un'unica CPU non è possibile eseguire più programmi contemporaneamente
- Si può simulare un esecuzione parallela, alternando rapidamente l'attività della CPU tra programmi differenti
  - *Context switching*
  - Normalmente, il sistema operativo si prende cura di tutti i dettagli di tale alternanza

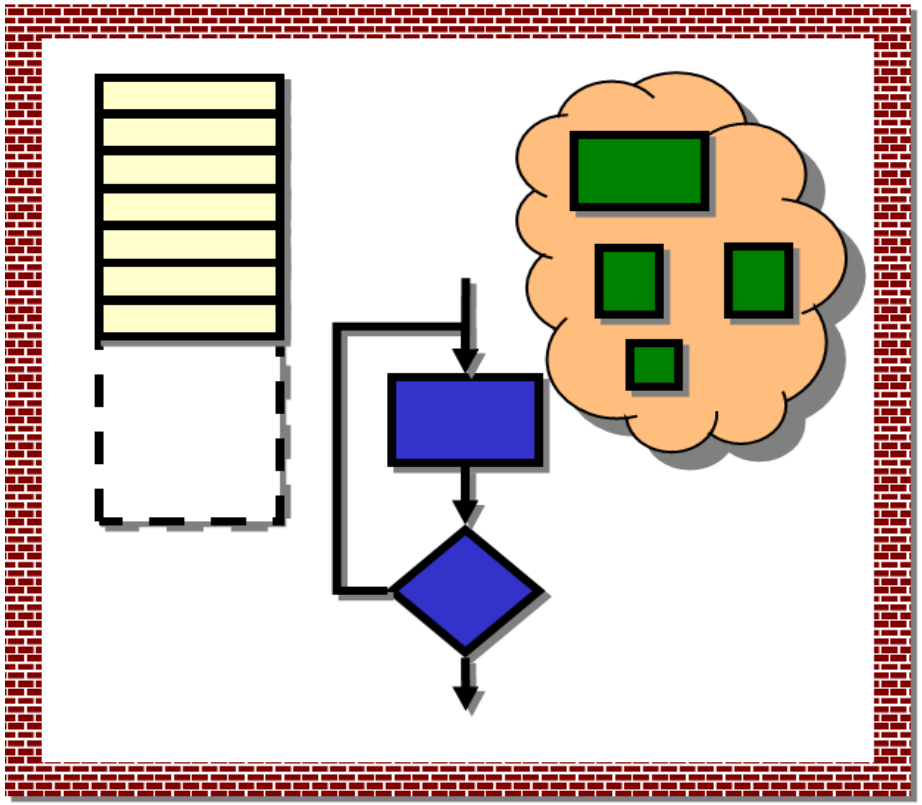
Simultaneità di esecuzione

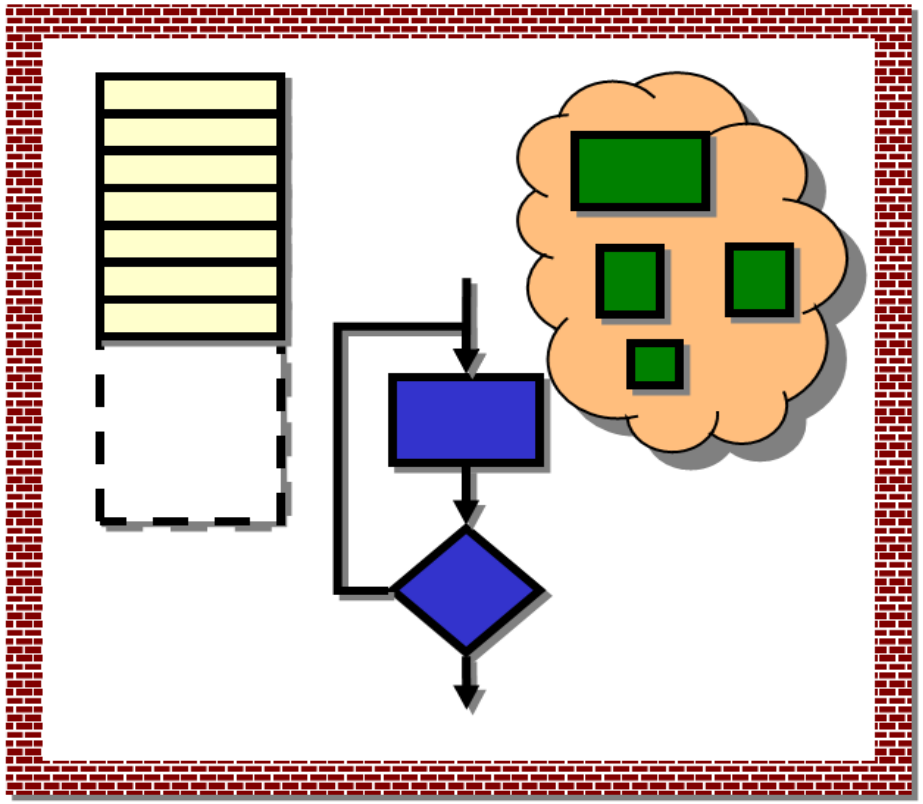


Esecuzione parallela

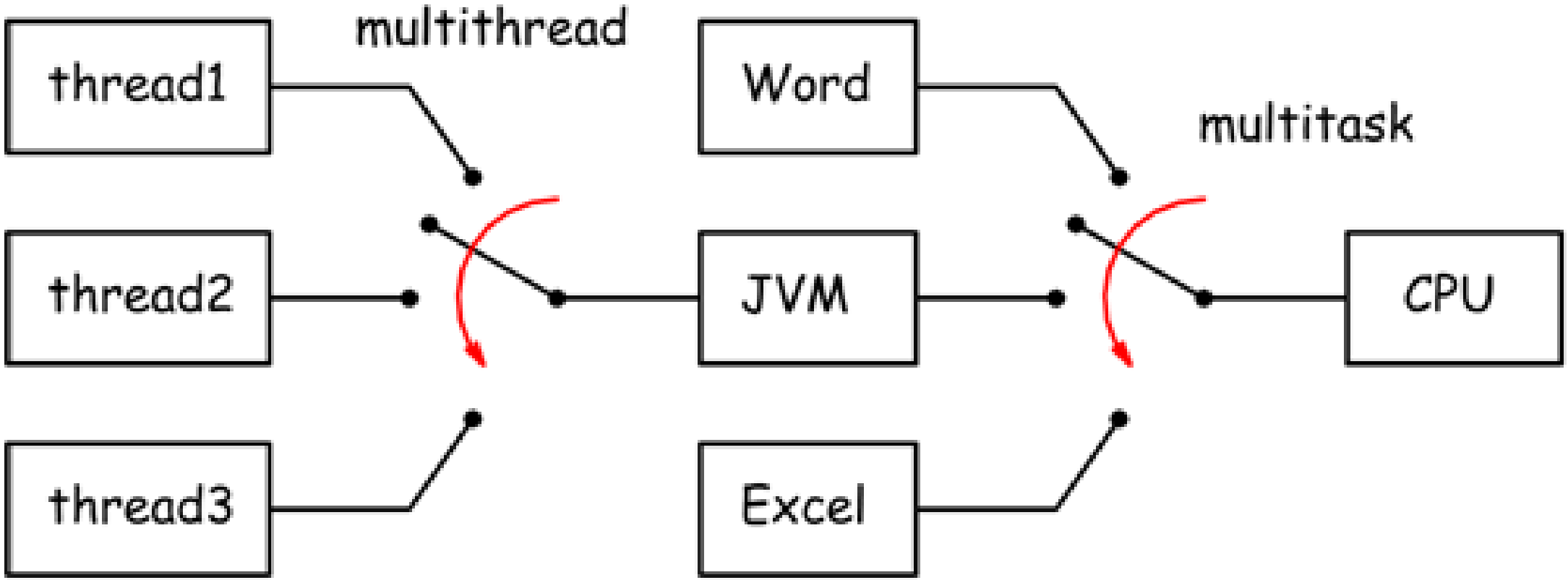
- Il parallelismo può avvenire
  - Tra processi differenti
  - All'interno di un dato processo, tra flussi di esecuzione differenti
- Nel primo caso, c'è una completa indipendenza di esecuzione
  - I programmi sono eseguiti in spazi di indirizzamento separati: le elaborazioni di uno non sono accessibili all'altro
  - Eventuali interazioni sono possibili solo tramite i servizi offerti dal sistema operativo (comunicazione tra processi)
- Nel secondo caso, i due flussi di esecuzione cooperano per il raggiungimento di un obiettivo comune
  - Condividendo lo spazio di indirizzamento e le informazioni in esso contenute

Processi paralleli





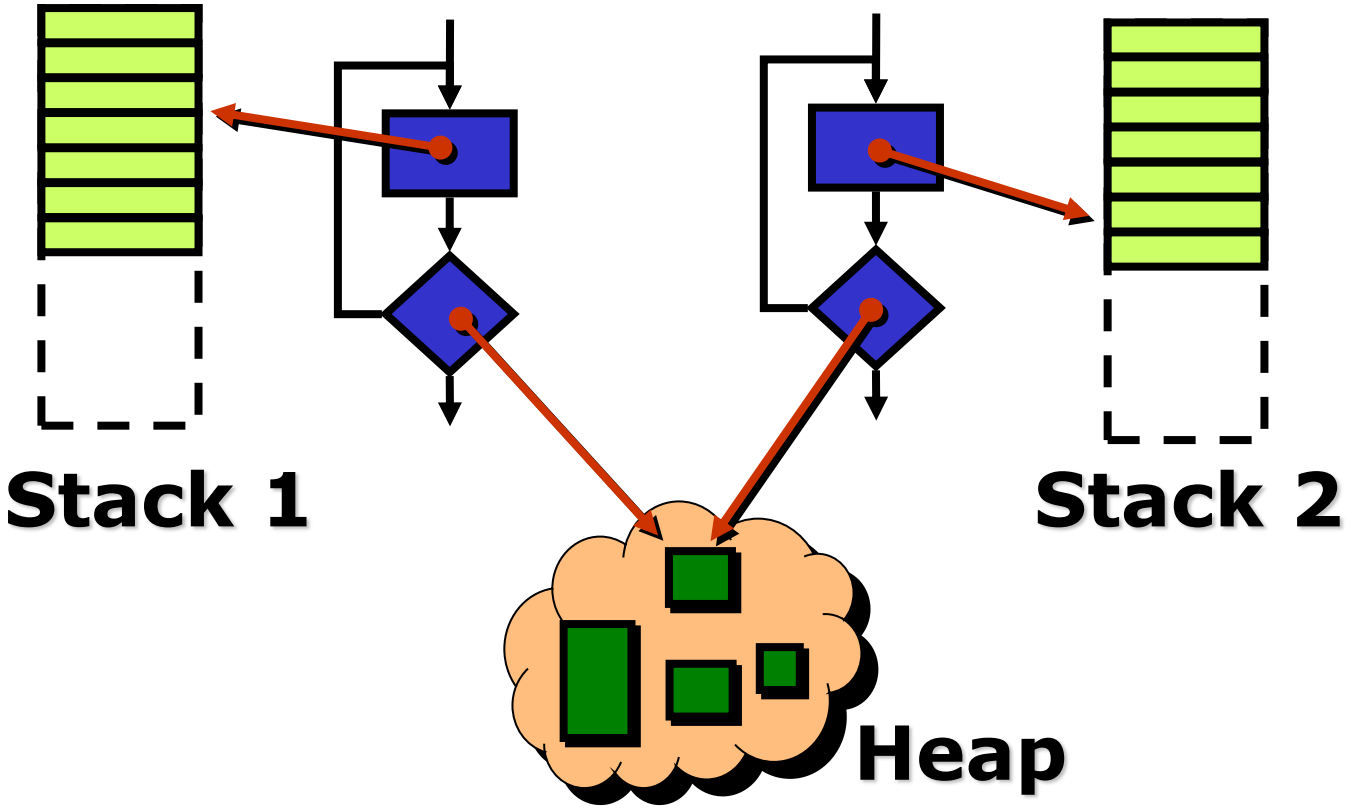
Sistema operativo


multithread
thread1
thread2
thread3
JVM
Word
Excel
multitask
CPU

Thread (1)

- Un thread è un singolo flusso di controllo sequenziale all'interno di un processo
  - All'atto della creazione del processo, il sistema operativo crea il thread principale ad esso associato
- Ad ogni thread è associato un "program counter", uno stack (con il relativo puntatore) ed altre informazioni di controllo
  - Ciascun thread si "dipana" secondo le normali regole di esecuzione (chiamate a funzioni, ritorni, gestione delle eccezioni, …)
- Tutti i thread di un dato processo condividono il codice ed i dati su cui operano
  - Il sistema operativo alterna l'esecuzione dei diversi thread
  - In qualunque istante, un thread può essere "congelato" nello stato in cui si trova, in attesa che torni il proprio turno di esecuzione
  - Il programma si comporta "come se" i diversi flussi fossero attivi contemporaneamente

Thread (2)



Quando si usano i thread...

- Per aumentare le prestazioni…
  - …se il programma è vorace di CPU…
  - … e si dispone di un calcolatore multiprocessore…
  - … e il S.O. è in grado di sfruttare più processori!

Quando si usano i thread…

- Per evitare che il programma si blocchi in attesa di operazioni naturalmente lente…
  - È il caso delle operazioni di I/O sul disco o sulla rete
  - La disponibilità di risorse condivise facilita la comunicazione

Quando si usano i thread….

- Per ricevere eventi asincroni dal sistema operativo…
  - È il caso delle interfacce grafiche o di altre attività di sistema
  - Richiede l'uso della tecnica di programmazione "reattiva"

Quando si usano i thread…

- Per realizzare sveglie e temporizzatori…
  - A volte occorre reagire allo scorrere del tempo
  - Lo si può fare utilizzando un thread che “dorme” per un dato intervallo e poi invia un messaggio

Quando si usano i thread…

- Per realizzare programmi che eseguono attività indipendenti…
  - Si può semplificare la scrittura del programma (?!) e migliorare i tempi di risposta nei confronti dell'utente

L’altra faccia della medeglia

- La programmazione basata su più thread introduce:
  - Complessità
  - Sovraccarico computazionale
- Un programma multithread deve garantire
  - Correttezza (*safety*)
  - Equità di accesso (*fairness*)
  - …

Creazione di thread

- Per creare un thread occorre:
  - Preparare le strutture di supporto necessarie alla sua esecuzione
  - Indicare quale codice esso debba eseguire
- La classe `java.lang.Thread` si occupa del primo aspetto, il programmatore del secondo

Creazione di un thread

- Si specifica il codice da eseguire:
  - creando una sottoclasse di **`Thread`** che ridefinisca il metodo **`run()`**
  - passando al costruttore della classe Thread un oggetto che implementi l'interfaccia **`Runnable`**
- Istanziando la classe Thread o una sua sottoclasse si prepara l'esecuzione di un nuovo flusso di esecuzione
  - All'atto della creazione, questo non è ancora attivo
  - Lo diventa quando viene invocato il metodo **`start()`**

Creazione di un thread

Primo metodo: *Sottoclasse Thread*

- Dichiarare una classe (es. **HelloThread)** come sottoclasse di **Thread** (package: **java.lang**)
- Sovrascrivere il metodo **run()** della classe **Thread**
- Creare un oggetto della classe **HelloThread** ed invocare il metodo **start()** per attivarl

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

Creazione di un thread

Secondo metodo: *oggetto Runnable*

- Definire una classe (es. **HelloRun**) che implementa l'interfaccia **Runnable.** N.B. **Runnable** ha il metodo **run()**.
- Creare un oggetto della classe **HelloRun**
- Creare un oggetto di classe **Thread** passando un oggetto di classe **HelloRun** al costruttore **Thread()**. Attivare l'oggetto **Thread** con il metodo **start()**

```
public class HelloRun implements Runnable {
     public void run() {
          System.out.println("Hello from a thread!");
     }
     public static void main(String args[]) {
          (new Thread(new HelloRun())).start();
     }
}
```

Metodi classe thread

run:

- **contiene il codice del thread**

start:

- **t.start() → esegue il thread t**

getName / getPriority :

- **restituisce il nome o la priorità del thread**

setName / setPriority:

- **modifica del nome o della priorità del thread**

sleep:

- **sospende l'esecuzione del thread per *m* millisecondi (valore di m passato come argomento)**

yield:

- **sospende l'esecuzione del thread corrente consentendo l'esecuzione di altri thread *pronti* e a uguale priorità**

join:

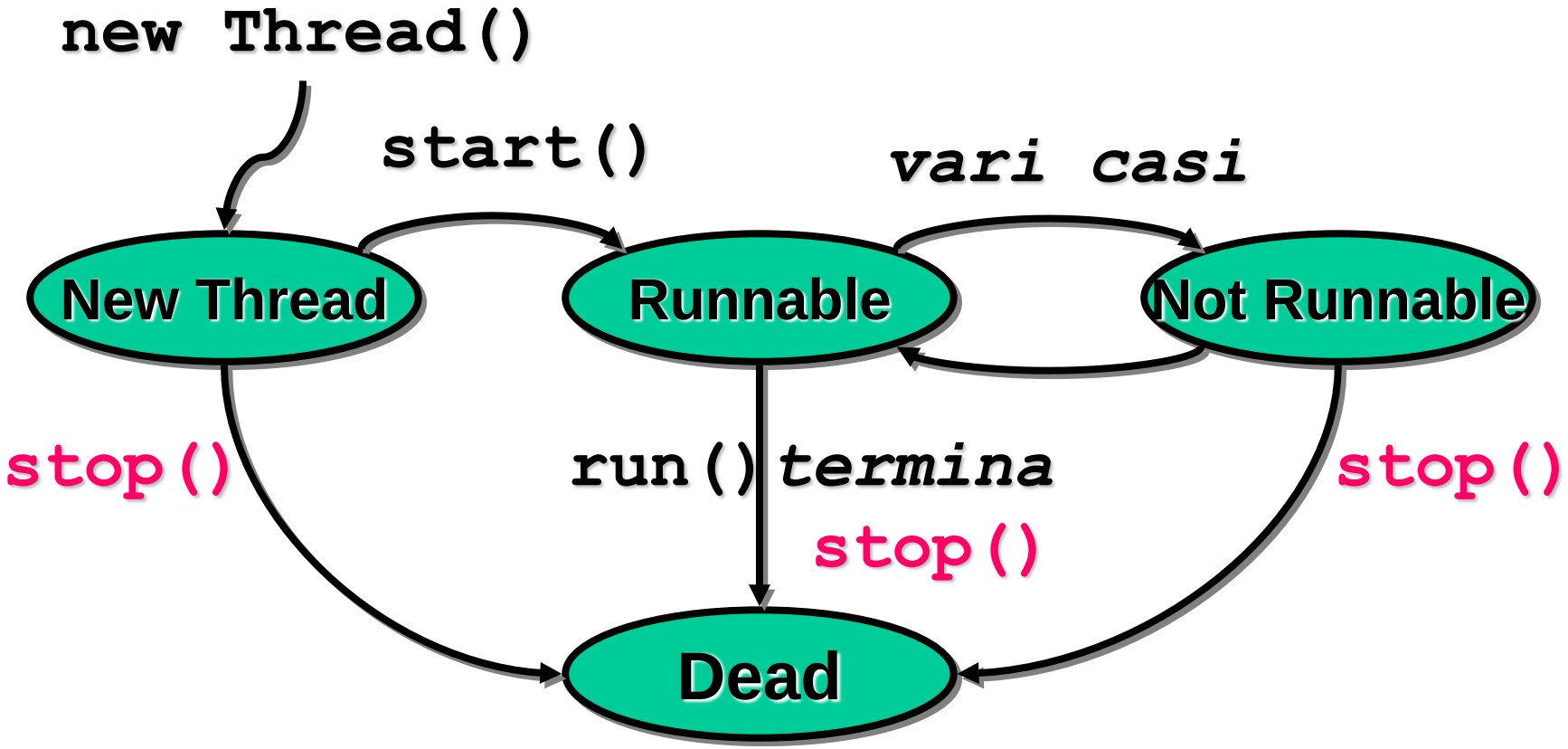
- **t.join() → attende la terminazione del thread t**

Thread concorrenti per accesso allo schermo

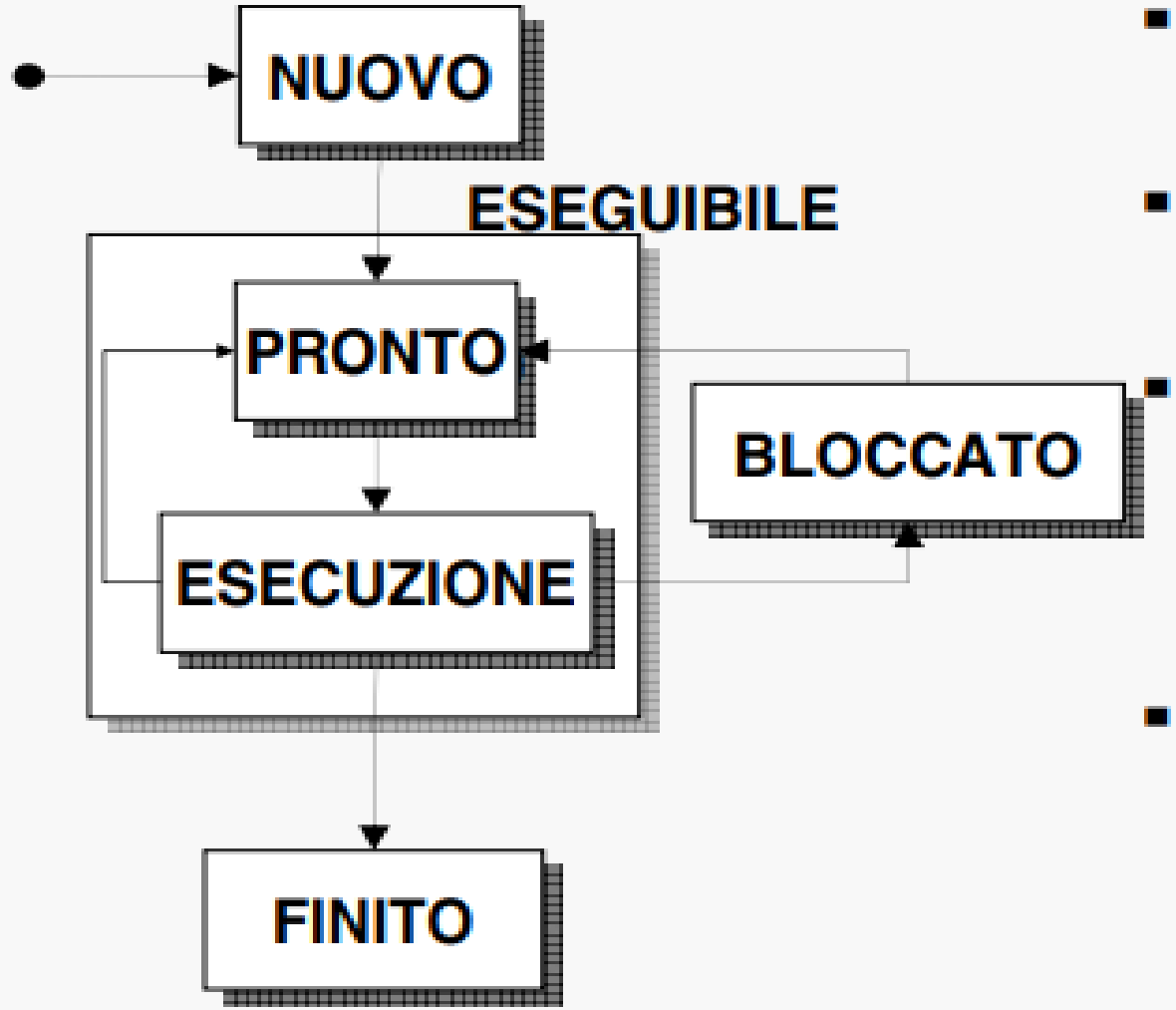
```
1  package thread;
2
3  public class ThreadEsempio1 {
4
5      public ThreadEsempio1() {
6
7      }
8
9      public static void main(String[] args) {
10         A T1 = new A();
11         B T2 = new B();
12         T1.start();
13         T2.start();
14     }
15
16     static class A extends Thread{
17         public void run() {
18             for (; ; ) {System.out.println("A");}
19         }
20
21     }
22     static class B extends Thread{
23         public void run() {
24             for (; ; ) {System.out.println("B");}
25         }
26
27     }
28 }
```

```java
package thread;

/*
 * Thread con stessa sottoclasse di Thread
 */
public class ThreadEsempio2 {

	public static void main(String[] args) {
		P T1 = new P('A');
		P T2 = new P('B');
		T1.start();
		T2.start();
	}

	static class P extends Thread{
		private char ch;

		public P(char c) {ch = c;}

		public void run() {
			for (; ; ) {System.out.println(ch);}
		}
	}
}

```

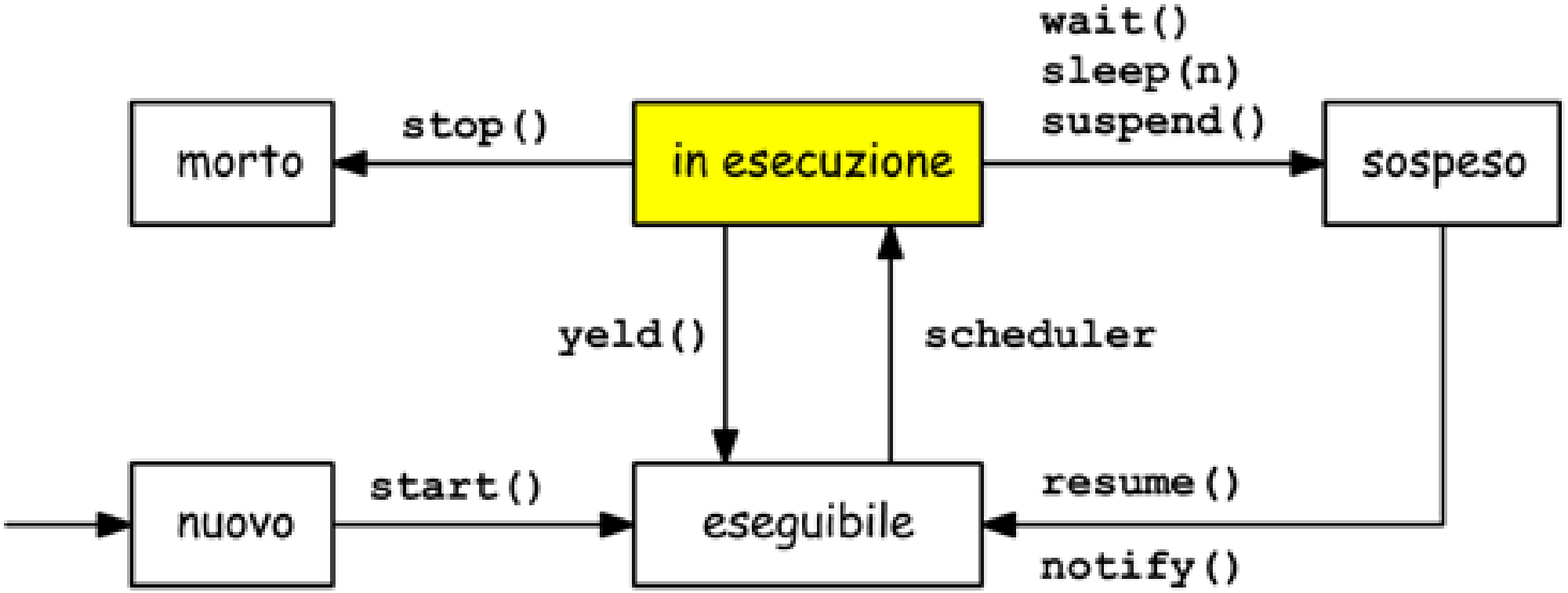
Stato di un thread



Stati di un thread



- **In esecuzione**: sta utilizzando la CPU
- **Pronto**: in attesa di utilizzare la CPU
- Gli stati **In esecuzione** e **Pronto** sono **logicamente equivalenti**: i processi sono **Eseguibili**
- **Bloccato**: in attesa di un evento. Un processo bloccato **non è eseguibile** (anche se la CPU è libera)


wait()
sleep(n)
suspend()
morto
stop()
in esecuzione
sospeso
yeld()
scheduler
nuovo
start()
eseguibile
resume()
notify()

Scheduling

- Ad ogni istante, la JVM sceglie tra tutti i thread nello stato Runnable quello/i che deve essere eseguito
  - Il numero di thread selezionati è pari al numero di processori effettivamente utilizzabili dalla JVM
  - L'esecuzione del thread selezionato procederà per un dato intervallo, fino a che non si verifica un opportuno evento
- Ad ogni thread è associata una priorità che viene utilizzata in questo processo di scelta:
  - Viene attribuita dal programmatore con il metodo `setPriority()`
  - Lo scheduler non la cambia
  - Se più thread sono eseguibili, viene scelto quello alla massima priorità
  - Se ci sono più thread alla massima priorità, ne viene selezionato uno a turno

Passaggio di stato

- Un thread diventa non eseguibile nei seguenti casi:
  - `sleep(long time)`
  - `obj.wait()`
  - esegue un'operazione di I/O bloccante
- Ritorna eseguibile se
  - trascorre il tempo
  - un altro thread chiama `obj.notify()`
  - finisce l'operazione di I/O
  - un altro thread esegue `t.interrupt()`

Conoscere lo stato di un thread

- Il metodo `isAlive()`, invocato su un oggetto di tipo Thread,
  - ritorna `true` se il thread è iniziato e non è ancora terminato
- Non è possibile determinare se un dato thread sia o no eseguibile
  - né, tanto meno, sapere se sia o no in esecuzione

Terminare un thread

- Un thread termina quando il metodo `run()` ritorna…
- …oppure se qualcuno ne invoca il metodo `stop()`
  - Alternativa “deprecata” perché può dare origine a problemi di sincronizzazione
- Se il thread contiene un ciclo conviene usare una variabile condivisa:
  - Ad ogni ripetizione del ciclo la variabile viene testata per decidere se è il caso di continuare oppure no
  - Occorre dichiarare tale variabile con il modificatore `volatile`, per evitare ottimizzazione errate da parte del compilatore

Terminare un thread (2)

```
private volatile Thread runner;

public void go() {
     runner= new Thread(this);
     runner.start();
}

public void halt() { runner=null; }

public void run() {
  Thread me;
  me = Thread.getCurrentThread();
  while (runner == me) {
    //esegui il ciclo
    //...
  }
}
```

```java
package thread;

public class ThreadEsempio3 {

    public static void main(String[] args) {
        P T1 = new P('A');
        P T2 = new P('B');
        T1.start();
        T2.start();
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        T1.stop();
        T2.stop();
        System.out.println("FINE");

    }

    static class P extends Thread{
        private char ch;
        public P(char c) {ch = c;}
        public void run() {
            for (; ; ) {System.out.println(ch);}
        }
    }
}
```

```java
package thread;
public class ThreadEsempio4 {
    public static void main(String[] args) {
        P P1 = new P('A');
        P P2 = new P('B');
        Thread T1=new Thread(P1);
        Thread T2=new Thread(P2);
        T1.start();
        T2.start();
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        P1.ferma();
        P2.ferma();
        System.out.println("FINE");
    }
    static class P implements Runnable{
        private char ch;
        boolean stop = false;
        public P(char c) {ch = c;}
        public void run() {
            for (; ; ) {
                System.out.println(ch);
                if (stop) {break;}
            }
        }
        public void ferma() {stop=true;};
    }
}
```

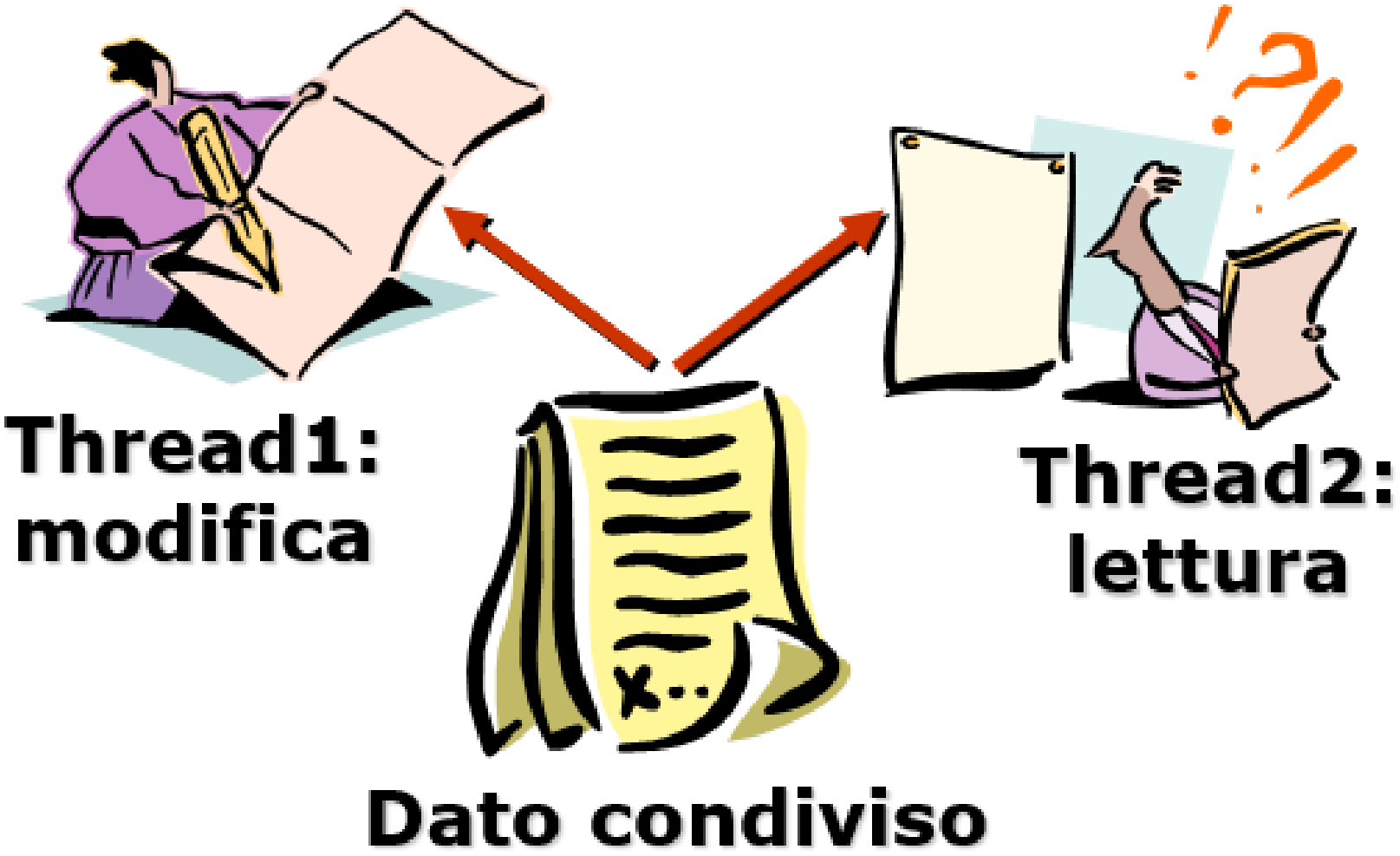
Attendere la morte di un thread

- Siano dati due thread, `t1` e `t2`…
- `t1` può attendere la terminazione di `t2` invocando `t2.join()`
- L'attesa di `t1` può essere interrotta chiamando `t1.interrupt()`

```java
package thread;

/*
 *  Si attiva Thread1
 *  Si aspetta che finisca thread1
 *  Si attiva Thread2
 *  Si aspetta che finisca thread2
 *  Si attiva Thread3
 *
 */
public class ThreadEsempio5 {

    public static void main(String[] args) {
        // creating two threads
        ThreadJoining t1 = new ThreadJoining();
        ThreadJoining t2 = new ThreadJoining();
        ThreadJoining t3 = new ThreadJoining();

        t1.setName("ThreadT1");
        t2.setName("ThreadT2");
        t3.setName("ThreadT3");

        // thread t1 starts
        t1.start();

        // starts second thread after when
        // first thread t1 has died.
        try
        {
            System.out.println("Current Thread: "
                    + Thread.currentThread().getName());
            t1.join();
        }

        catch(Exception ex)
        {
            System.out.println("Exception has " +
                "been caught" + ex);
        }

        // t2 starts
        t2.start();

        // starts t3 after when thread t2 has died.
        try
        {
            System.out.println("Current Thread: "
                    + Thread.currentThread().getName());
            t2.join();
        }

        catch(Exception ex)
        {
            System.out.println("Exception has been" +
                    " caught" + ex);
        }

        t3.start();
    }

}

class ThreadJoining  extends Thread{
    @Override
    public void run()
    {
        for (int i = 0; i < 2; i++)
        {
            try
            {
                Thread.sleep(2000);
                System.out.println("Current Thread: "
                        + Thread.currentThread().getName());
            }

            catch(Exception ex)
```

```
76              catch(Exception ex)
77              {
78                  System.out.println("Exception has" +
79                          " been caught" + ex);
80              }
81              System.out.println(i);
82          }
83      }
84  }
85
86
```

Thread: problemi introdotti



Concorrenza

- Due o più thread possono essere eseguiti contemporaneamente
  - Se operano su dati condivisi, questi dati possono modificarsi inaspettatamente
- In Java, solo gli oggetti (istanze della classe *Object)* sono condivisibili
  - Le variabili locali di tipo elementare (interi, caratteri, booleani,…) sono ospitate sullo stack, ogni thread ne mantiene una copia indipendente
  - Occorre identificare le porzioni di codice (metodi) che operano su tali dati e proteggerle opportunamente

Compiti del programmatore

- Java fornisce solo meccanismi semplici di sincronizzazione
  - È compito del programmatore riconoscere quando e dove utilizzarli
- Un uso sbagliato porta a risultati disastrosi…
  - Mancata sincronizzazione: malfunzionamenti casuali
  - Errata sincronizzazione: blocco

Correttezza

- “Non capita mai che”…
  - …un thread “operi” su un oggetto, alterandone il contenuto, mentre un altro sta già operando sullo stesso oggetto
  - Sui dati condivisi, occorre operare “uno alla volta”
- È una proprietà degli oggetti condivisi
  - Questi mantengono al proprio interno degli “invarianti” identificati dal programmatore

Sincronizzazione

- Mentre una modifica è in corso bisogna evitare che altri thread accedano alla risorsa condivisa:
  - È necessario introdurre un meccanismo che regoli l'accesso alle zone di codice potenzialmente pericolose
- Non è possibile realizzare un semaforo direttamente in Java:
  - diventerebbe, infatti, una risorsa condivisa e necessiterebbe di un semaforo…

Sezioni critiche

- Porzioni di codice che operano su dati condivisi:
  - Mentre un thread esegue una sezione critica, nessun altro thread dovrebbe poter agire sui dati condivisi
- Ciò si ottiene con il meccanismo di “mutua esclusione” fornito da Java
  - Utilizzo della parola chiave **synchronized**

Sezioni critiche

Sezione critica:

- *Sequenza di istruzioni che deve essere eseguita in modo **mutuamente esclusivo** con altre sezioni critiche*

Classe di sezioni critiche:

- *Insieme di sezioni critiche le cui esecuzioni devono essere **mutuamente esclusive***

Java identifica la sezione critica per mezzo della parola chiave **synchronized**

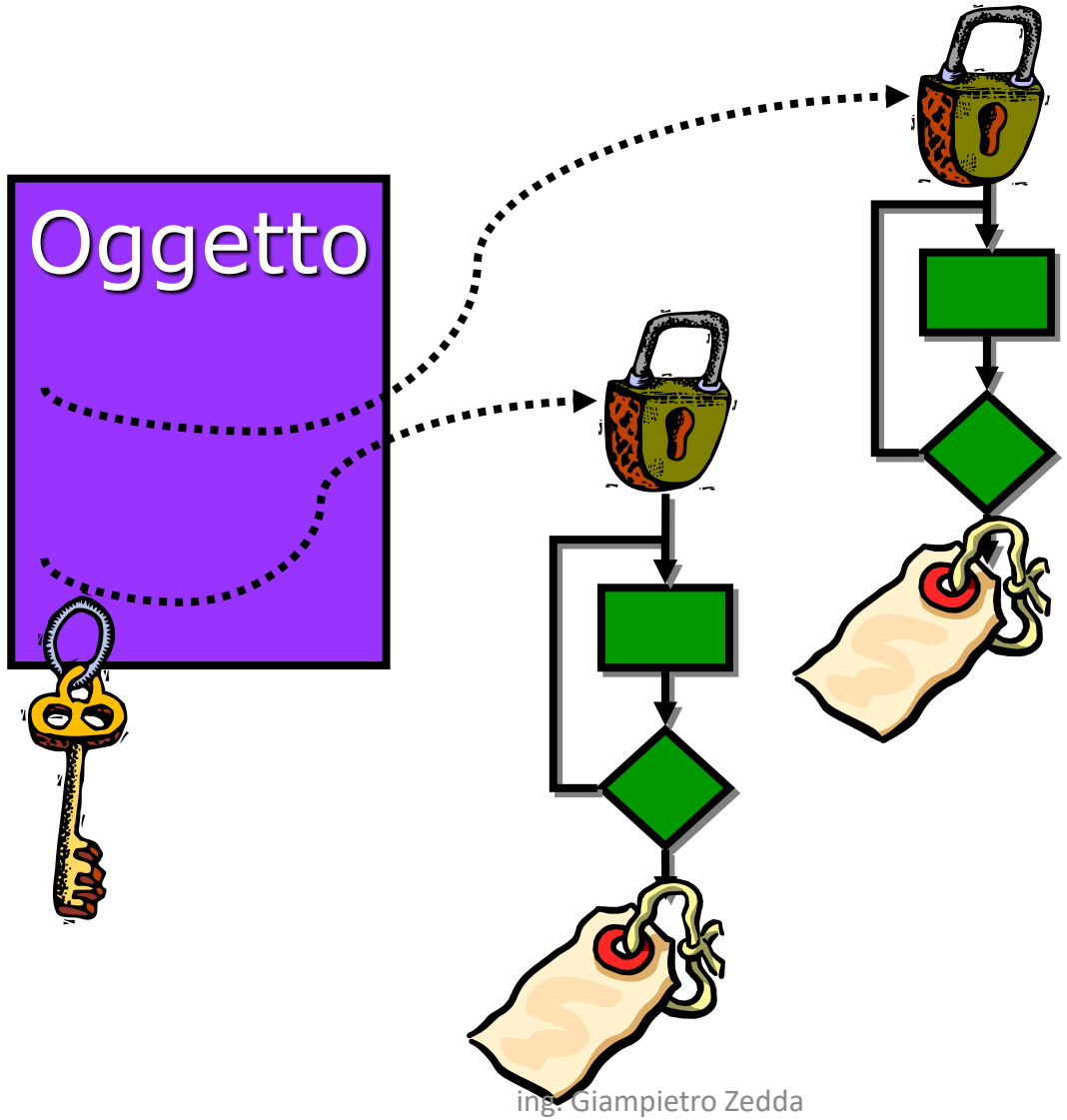
- Metodo sincronizzato
- Blocco sincronizzato

La mutua esclusione è realizzata per mezzo di un *lock*, o semaforo binario.

Mutua esclusione (1)

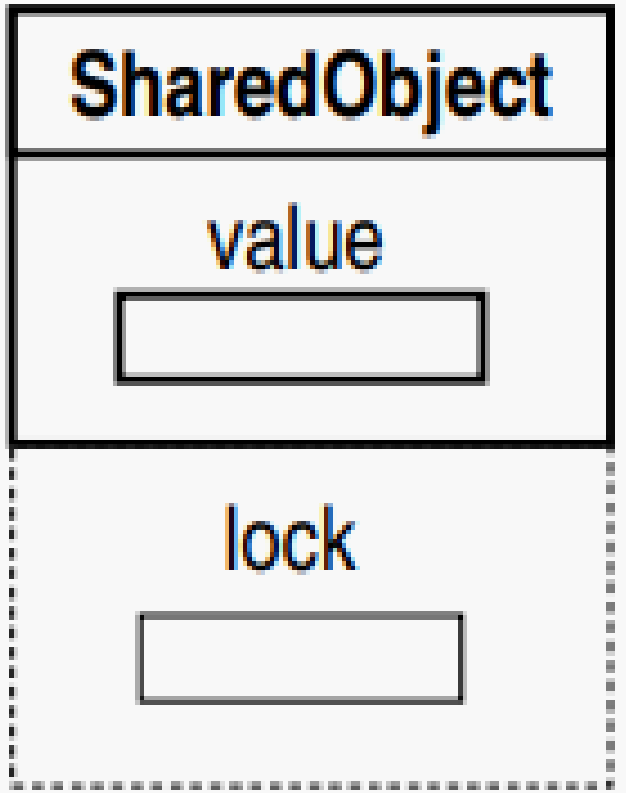
synchronized
metodo1()

synchronized
metodo2()



Implementazione semplificata

```
class SharedObject {
   private int value = 0;
   synchronized void write(int v) {
      value = v;
   }
   synchronized int read() {
      return value;
   }
}
```



Mutua esclusione (2)

- Ogni oggetto dispone di di un'unica "chiave" che controlla l'accesso ai metodi sincronizzati
  - Per poter eseguire il codice è necessario "aprire" il lucchetto
- Il primo thread che cerca di eseguire un metodo sincronizzato, prende la "chiave" e la tiene fino al termine del metodo stesso
  - Eventuali altri thread che cerchino di accedere oltre un lucchetto devono attendere la disponibilità della "chiave"
  - Nel caso in cui si verifichi un'eccezione durante l'esecuzione del metodo, la chiave viene comunque rilasciata

Esistenza in vita

- Ogni thread modella un'attività
  - Sequenza di operazioni che si svolgono ordinatamente secondo le regole del linguaggio di programmazione
- A volte, un'attività coinvolge operazioni naturalmente lente
  - Lettura da disco o da rete, interazione con l'utente, attesa di un evento esterno, …
- …altre volte, alcune operazioni potenzialmente "veloci" vengono rallentate per evitare conflitti sulla correttezza
  - Attesa che un altro thread esca da una sezione critica
- L'ordine con cui si effettuano tali operazioni può impedire la prosecuzione dell'attività nel suo complesso
  - Deadlock, livelock

Blocchi e metodi sincronizzati

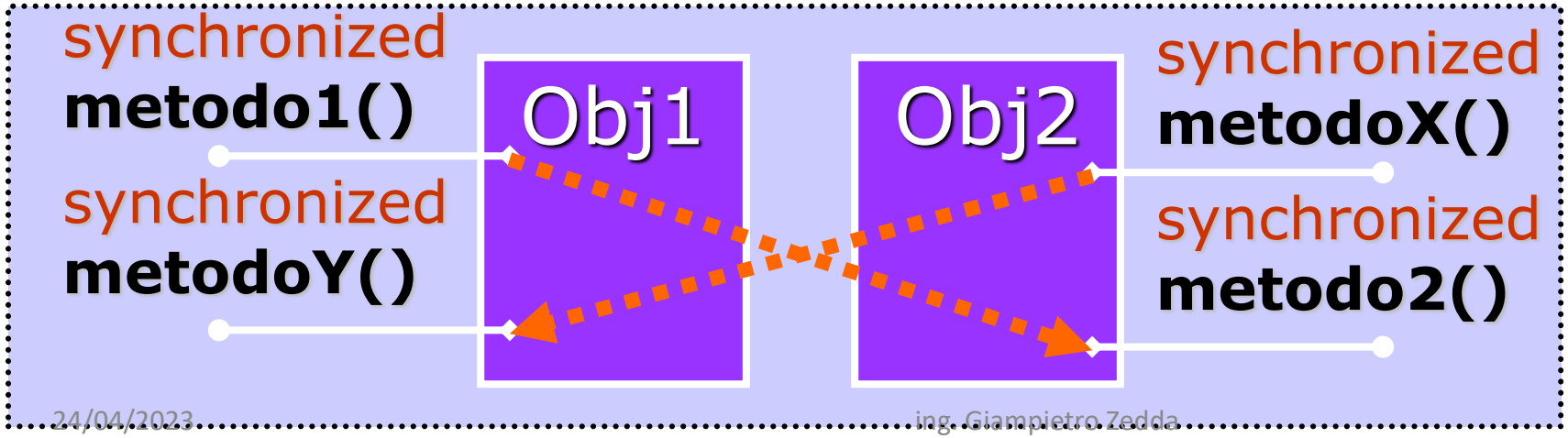
```
synchronized  void m(args) {
   /* sezione critica   */
}
```

è equivalente a

```
void m(args) {
   synchronized (this)
   { /* sezione critica */       }
}
```

Deadlock

- Un thread può trovarsi in più sezioni critiche allo stesso tempo:
  - chiamando un metodo sincronizzato di un oggetto dall'interno di un metodo sincronizzato di un altro oggetto
- Questo può dare origine a forme di blocco



Segnalazione tra thread

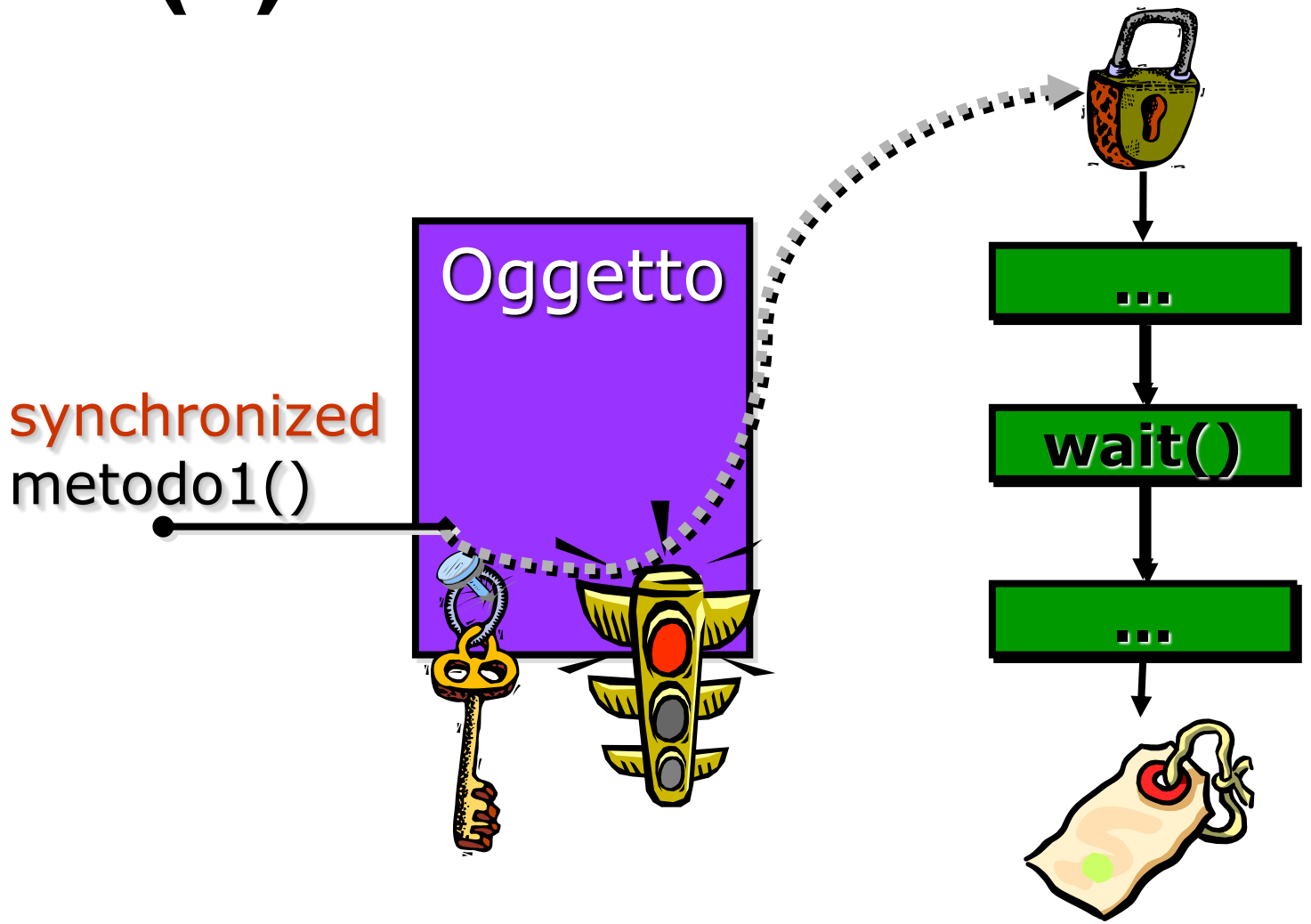
- A volte, per proseguire,
  un thread deve attendere
  il verificarsi di una
  data condizione
  in un altro thread
  - Occorre evitare di perdere tempo/risorse interrogando continuamente la condizione stessa in un ciclo attivo
- In java si realizza questo comportamento con la coppia di metodi **`wait()`**/**`notify()`**

Wait (1)

- Ogni oggetto dispone di un semaforo inizialmente bloccato
- Si attende il via libera con il metodo **`wait()`**
- Occorre invocare tale metodo solo all'interno di codice sincronizzato

```
synchronized (this) {
    wait();
}
```

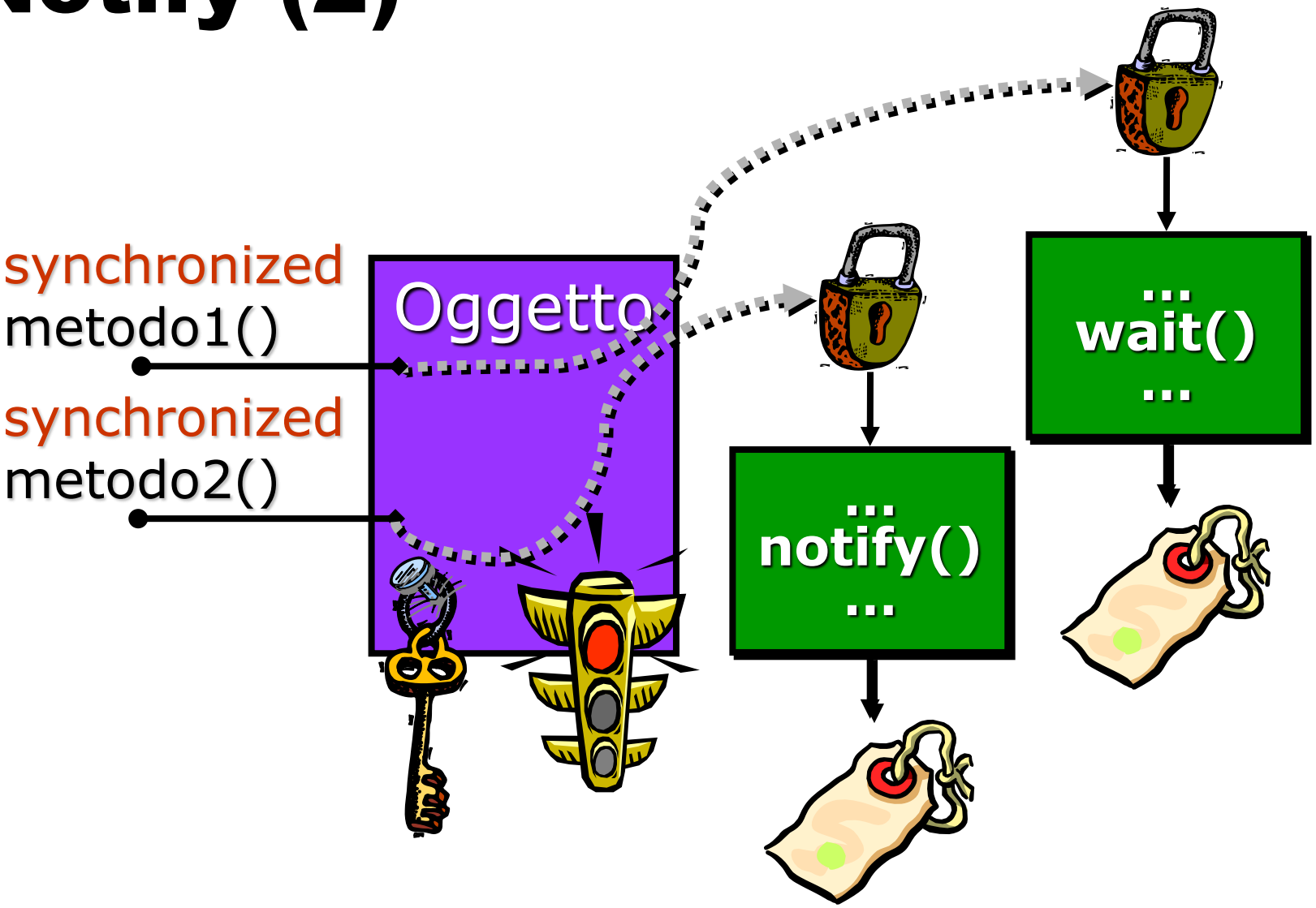
Wait (2)



Notify (1)

- Fornisce il “via libera” ad un thread in attesa tramite wait()
- Se nessuno è in attesa non capita nulla
- Se più di un thread è in attesa, ne viene scelto uno a caso
- Si può chiamare questo metodo all’interno di codice sincronizzato:
  - Poiché sia il thread che invia la notifica che quelli in attesa sono all’interno di una sezione critica, solo quando chi ha effettuato la notifica esce dal blocco sincronizzato, il thread svegliato può iniziare

Notify (2)


synchronized
metodo1()
synchronized
metodo2()
Oggetto
...
wait()
...
...
notify()
...

NotifyAll

- notifyAll() sveglia tutti i thread in attesa su un dato semaforo
- Solo quando il thread che ha inviato la notifica rilascia la chiave, gli altri metodi possono continuare l'esecuzione (uno per volta, fino al rispettivo rilascio)

Meccanismi di comunicazione

- Il meccanismo wait/notify si limita ad indicare che il thread in attesa può continuare
- Se occorre anche scambiare dei messaggi tra i due thread è necessario ricorrere all'uso di oggetti condivisi

Meccanismi di comunicazione

- La complessità di gestione della sincronizzazione spinge verso l'uso di tecniche note ed affidabili (Pattern)
- È sempre necessaria molta cautela nella realizzazione di programmi multi-thread

Il percorso

- Risorse condivise
- Il problema del Produttore e del Consumatore
- Readers & Writers

Accesso alle risorse condivise

- Se la risorsa da proteggere è tutta contenuta in un oggetto…
  - …l'uso di metodi sincronizzati può risultare sufficiente
- A volte, però, occorre proteggere risorse che si trovano in oggetti differenti:
  - Serve una soluzione più elaborata

La classe ControllaRisorsa

- Dispone dei metodi:
  - void assumiControllo()
  - void rilasciaControllo()
- Il programmatore si deve fare carico dell'utilizzo corretto:
  - Prima di accedere alla risorsa, occorre assumerne il controllo
  - Quando si non accede più, occorre rilasciarlo
  - La risorsa protetta può essere una collezione arbitrariamente complessa

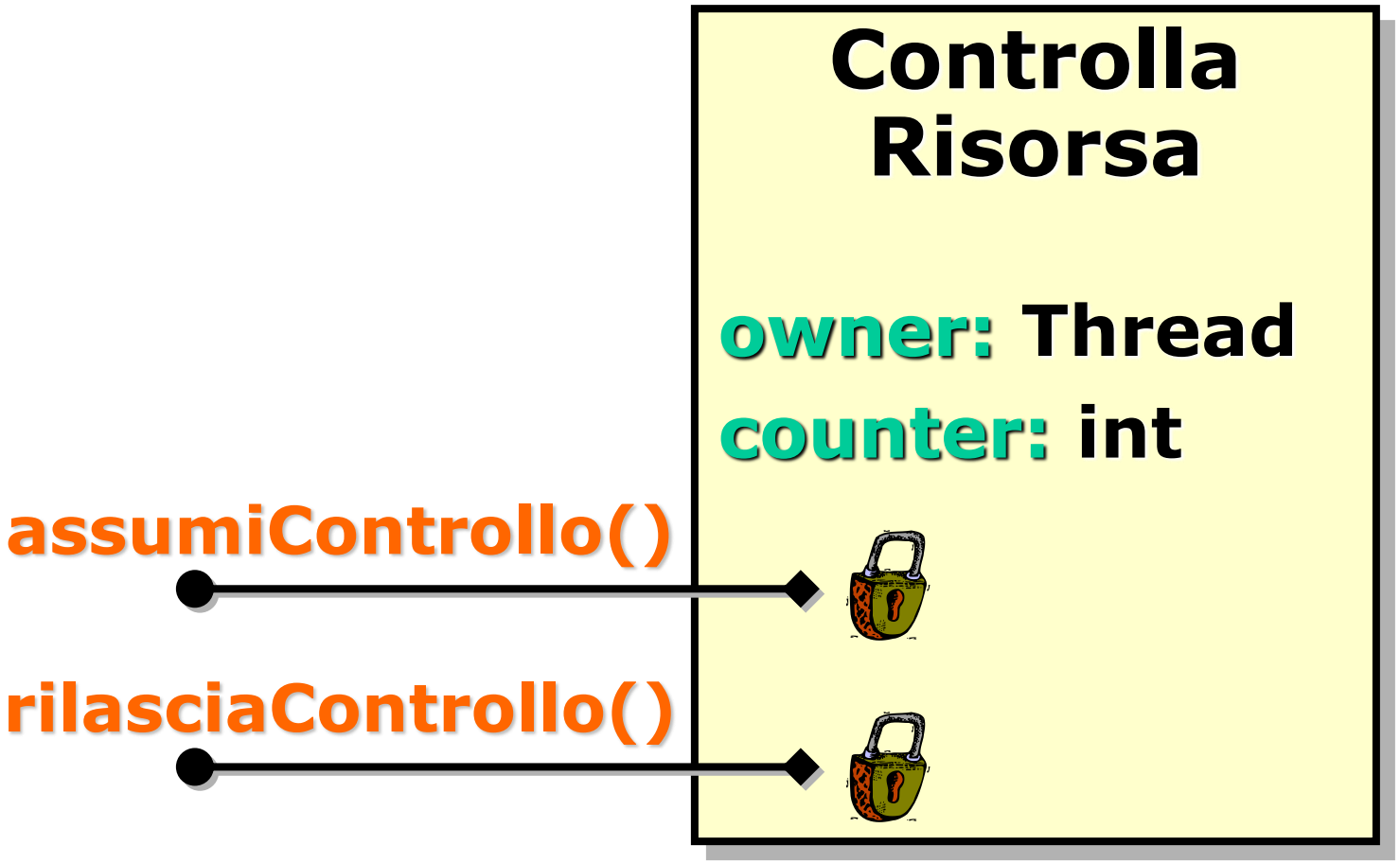
ControllaRisorsa (1)

```
public class ControllaRisorsa {
  private volatile Thread owner=null;
  private volatile int counter=0;

  public synchronized void assumiControllo() {
    while (! tentaAssunzione()) {                  //prova a prendere il controllo
      try {
        wait();                                    // se non riesce, aspetta
      } catch (InterruptedException ie) { }
    }
  }

  public synchronized void rilasciaControllo() {
    if (owner==Thread.currentThread()) {           //se possiede il controllo
      counter--;                                   //decrementa il contatore
      if (counter==0) {                            //se la risorsa è stata rilasciata
                                                   //tante volte quante richiesta
        owner=null;                                //rilascia il controllo
        notifyAll();                               //sveglia i thread in attesa
      }
    }
  }                                          //...continua
```

ControllaRisorsa (2)

```
  private synchronized boolean tentaAssunzione() {
    Thread me=Thread.currentThread();

    if (owner==null) {          // Nessuno ha il controllo,
      owner=me;                 // il thread corrente lo può assumere
      counter=1;
      return true;
    }
    else if (owner==me) {       // Il thread ha gia' il controllo,
      counter++;                // incrementa il contatore
      return true;
    } else
      return false;             // Risorsa occupata
  }


} //fine della classe
```

ControllaRisorsa



Vantaggi e svantaggi

- Soluzione concettualmente semplice ma di basso livello
- Bisogna ricordarsi di chiamare entrambi i metodi
- Nessuna relazione tra la risorsa controllata e l'oggetto che la controlla

PRODUTTORE E CONSUMATORE 1

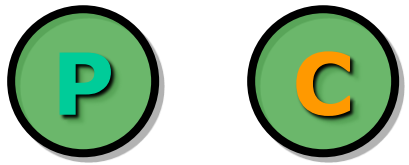
```java
package thread;
/*
 * Produttore consumatore
 */
public class ThreadEsempio7 {
    public static void main(String[] args) {
        OggettoCondviso c = new OggettoCondviso();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 2);
        p1.setName("Thread Producer");
        c1.setName("Thread Receiver");
        p1.start();
        c1.start();
    }
}

class OggettoCondviso {
    private int contents;
    private boolean available = false;

    synchronized int get() {
        while (available == false) {
            try {
                wait();
            }
            catch (InterruptedException e) {
            }
        }
        available = false;
        notifyAll();
        return contents;
    }

    synchronized void put(int value) {
        while (available == true) {
            try {
                wait();
            }
            catch (InterruptedException e) {
            }
        }
        contents = value;
        available = true;
        notifyAll();
    }
}

/*
 * Consumatore
 */
class Consumer extends Thread {
    private OggettoCondviso oggettoCondiviso;
    private int number;
    public Consumer(OggettoCondviso c, int number) {
        oggettoCondiviso = c;
        this.number = number;
    }
    public void run() {
        int value = 0;
        for (int i = 0; i < 50; i++) {
            value = oggettoCondiviso.get();
            System.out.println(Thread.currentThread().getName() + " Numero " + this.number+ " get: " + value);
        }
    }
}

/*
```

PRODUTTORE E CONSUMATORE 2

```
67 /*
68  * Produttore
69  */
70 class Producer extends Thread {
71     private OggettoCondviso oggettoCondiviso;
72     private int number;
73 
74     public Producer(OggettoCondviso c, int number) {
75         oggettoCondiviso = c;
76         this.number = number;
77     }
78 
79     public void run() {
80         for (int i = 0; i < 50; i++) {
81             oggettoCondiviso.put(i);
82             System.out.println(Thread.currentThread().getName() + " Numero " + this.number + " put: " + i);
83             try {
84                 sleep((int)(Math.random() * 100));
85             } catch (InterruptedException e) { }
86         }
87     }
88 }
```

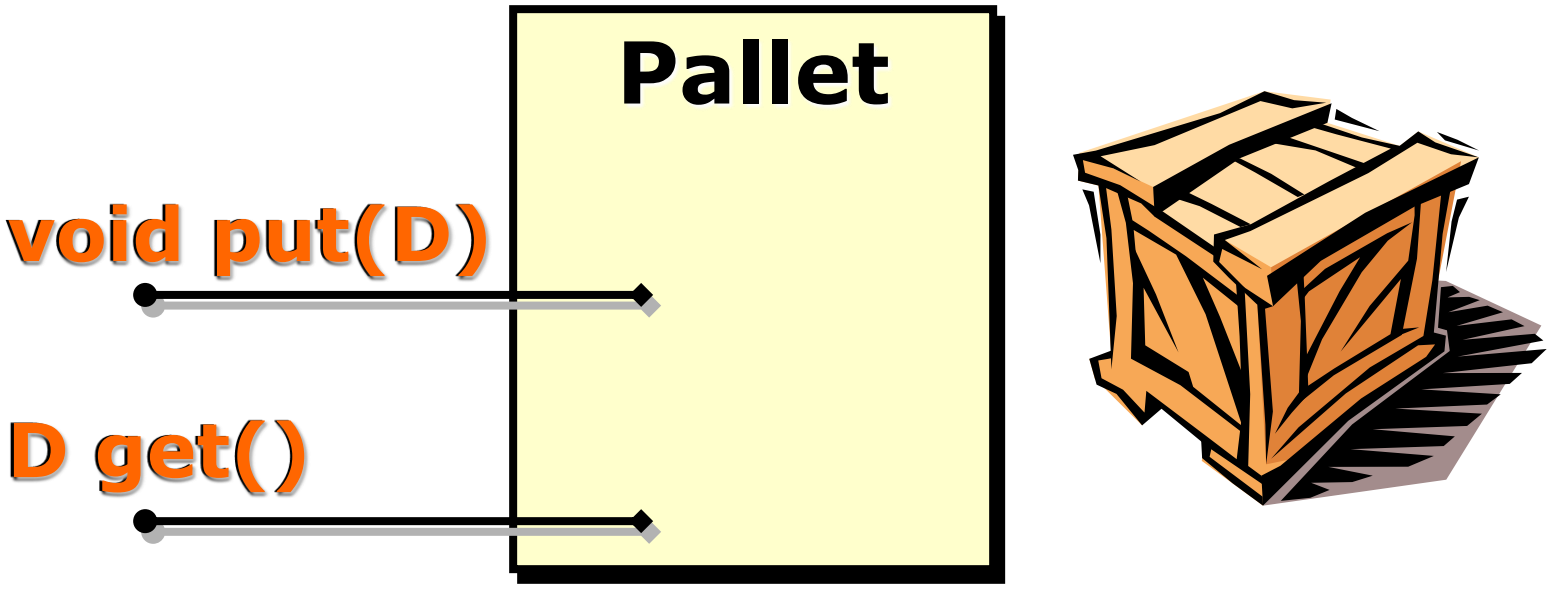
Produttore e consumatore

- Due thread:

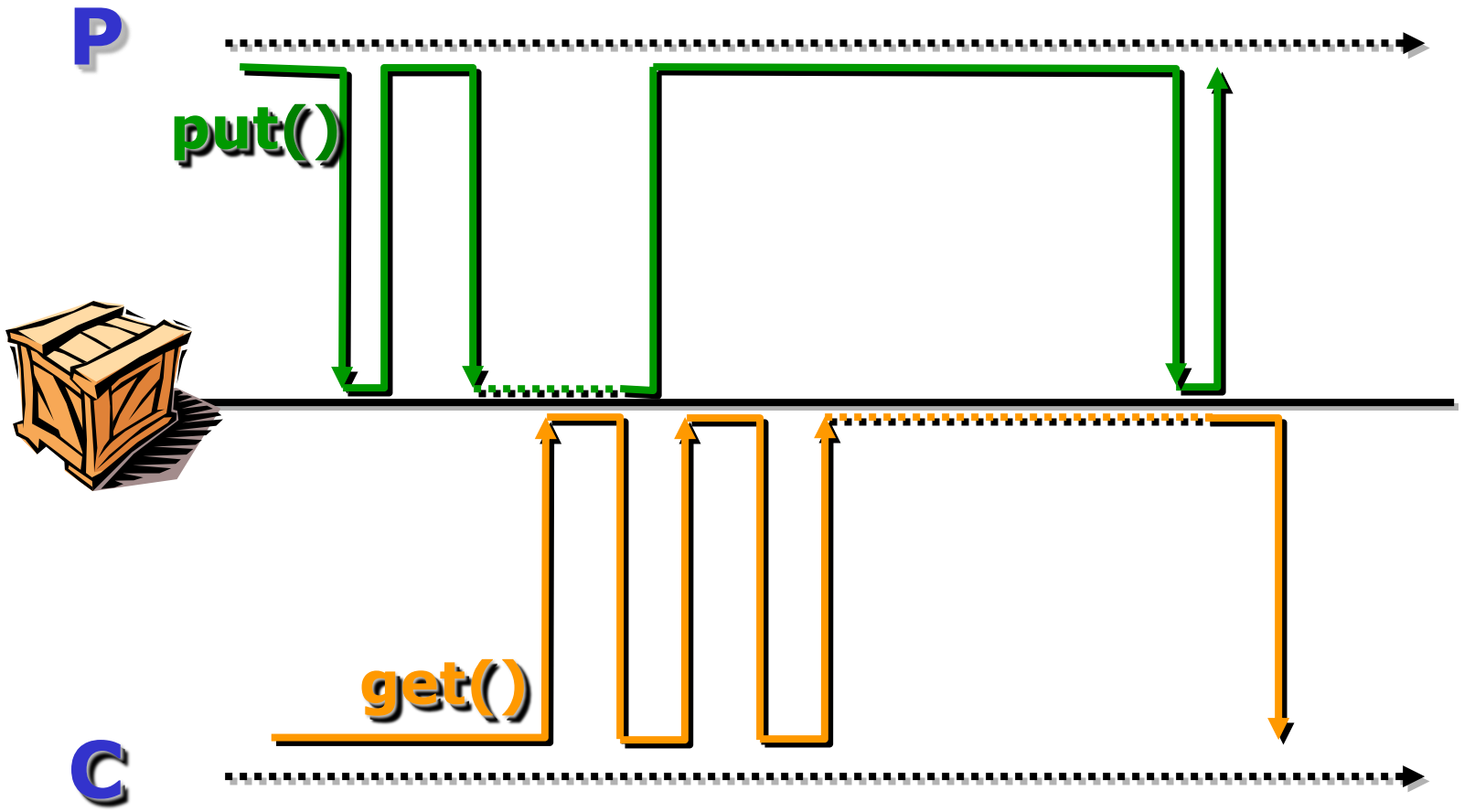


  - P produce continuamente oggetti e non può essere fermato
  - C riceve gli oggetti inviati da P e li elabora
- Vincoli ulteriori
  - Nessun oggetto prodotto deve andare perso
  - Ogni oggetto deve essere elaborato una sola volta dal consumatore

La soluzione



Produttore e consumatore



Pallet

```
public class Pallet {
  private Object dato=null;
  private boolean datoPresente= false;

  public synchronized Object get()  throws InterruptedException {
        while (! datoPresente) wait();
        datoPresente = false;
        Object o=dato;
        dato=null;
        notifyAll();
        return o;
   }

   public synchronized put(Object dato) throws InterruptedException {
        while (datoPresente) wait();
        this.dato = dato;
        datoPresente = true;
        notifyAll();
   }
} //fine della classe
```

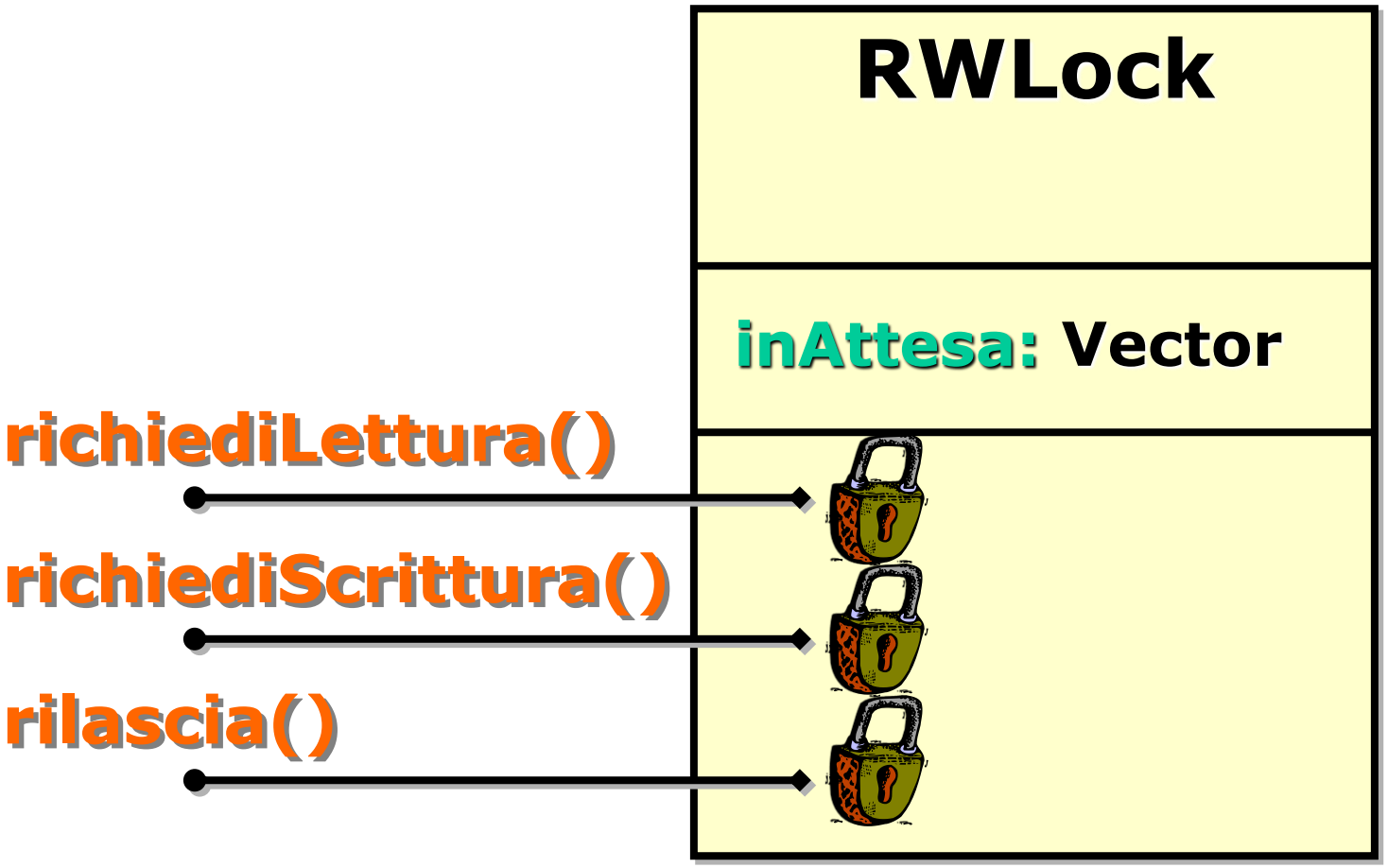
Vantaggi e svantaggi

- Soluzione elegante e affidabile
- Leggere variazioni (notify() al posto di notifyAll()) possono condurre a risultati inattesi

Readers & Writers (1)

- Data una risorsa condivisa:
  - possono esserci contemporaneamente molti accessi in **lettura**
  - ma un solo accesso in **scrittura** per volta
- Finché ci sono solo richieste di lettura, queste vengono accordate…
- Se arriva una richiesta di scrittura:
  - si attende che non ci siano altre attività in corso
  - dopodiché viene accordata
- Le ulteriori richieste di lettura giunte dopo una data richiesta di scrittura possono essere evase solo al termine di questa

Readers & Writers (2)



Problemi implementativi

- È necessario gestire il caso in cui uno stesso thread richieda l'accesso più volte
- Due sottocasi:
  - Stesso livello di privilegio
  - differente privilegio (lock upgrade)

Esempio di realizzazione

- RWLock.java
- RWLockTest.java

Vantaggi e svantaggi

- Pattern di utilizzo frequente
- Nessun collegamento tra la risorsa protetta e l'oggetto che la controlla

Principi generali sull'uso dei Thread

- Sincronizzare l'accesso ai dati condivisi
- Evitare di chiamare metodi *synchronized* all'interno di altri metodi *synchronized*
  - Può dare origine al blocco del programma
- Ridurre il tempo in cui si detiene l'accesso esclusivo ad una data risorsa
  - Per non rallentare altri thread
- Studiare attentamente i metodi invocati all'interno di un blocco sincronizzato
  - Documentare opportunamente il codice che si produce rispetto all'esecuzione concorrente