Corso di Programmazione Concorrente

Java Thread

Valter Crescenzi

http://crescenzi.inf.uniroma3.it

Sommario

- Java Thread e supporto alla PC della piattaforma Java
- Creazione di Thread con `java.lang.Thread` e `Runnable`
  - `join()` di Thread
- Interruzione di Thread e terminazione cooperativa
- Competizione in Java nativo
- Cooperazione in Java nativo
  - Monitor vs Java Monitor
  - Produttori/consumatori su buffer unitario
  - Risoluzione dei problemi di competizione
  - Risoluzione dei problemi di cooperazione
- TSD

Thread

- Flusso di esecuzione che condivide lo spazio di indirizzamento con gli altri thread
- Differenze rispetto ai processi:
  - le uniche risorse specifiche del thread sono quelle necessarie a garantire un flusso di esecuzione indipendente (contesto, stack, …)
  - tutte le altre risorse, ivi incluso lo spazio di indirizzamento, sono condivise con gli altri thread

Java Thread

- I thread in java sono stati supportati sin dalle primissime versioni della piattaforma in maniera *nativa*
- Tuttavia il supporto è stato oggetto di successive revisioni, anche importanti
  - affidando la gestione della concorrenza nella piattaforma java ad esperti del settore
- Per questo motivo esistono diversi *strumenti* per la PC, e taluni sostituiscono integralmente altri
- Questo testimonia quanto sia considerata importante il supporto alla PC per la piattaforma Java
  - Ma ha anche generato motivi di confusione

Thread & JVM

- In realtà qualsiasi programma java è multithread
  - Il garbage-collector è gestito da un thread separato
- Se si usano GUI, la JVM crea almeno due altri thread
  - Uno per gestire gli eventi della GUI
  - Uno per il rendering grafico

...nelle successive versioni della JVM il numero di thread iniziale è stato incrementato

Thread in Java: Creazione e Terminazione

- Creazione di Thread
  - si creano oggetti istanze della classe `java.lang.Thread`.
    - Il thread viene effettivamente creato dalla JVM non appena si richiama il metodo `start()`
    - Il nuovo thread esegue il metodo `run()`
- Terminazione di Thread
  - i thread terminano quando
    - finisce l'esecuzione del metodo `run()`
    - sono stati interrotti con il metodo `interrupt()`
    - eccezioni ed errori
- Join con Thread
  - richiamando il metodo `join()` su un oggetto Thread si blocca il thread corrente sino alla terminazione del thread associato a tale oggetto

`java.lang.Thread`

- I Thread della JVM sono associati ad istanze della classe `java.lang.Thread`
- Gli oggetti istanza di tale classe svolgono la funzione di interfaccia verso la JVM che è l'unica che può effettivamente creare nuovi thread
- Attenzione a non confondere il concetto di thread con gli oggetti istanza della classe `java.lang.Thread`
  - tali oggetti sono solo lo strumento con il quale è possibile *comunicare* alla JVM richieste
    - di creazione di nuovi thread
    - di interruzione dei thread esistenti
    - di attendere la fine di un thread (join)
    - …

2 Modi per Creare Thread in Java

- Instanziare classi che derivano da `java.lang.Thread`
  - più semplice
  - ma non è possibile derivare da altre classi
- Instanziare direttamente `Thread` passando al suo costruttore un oggetto di interfaccia `Runnable`
  - bisogna implementare l'interfaccia `Runnable` e creare esplicitamente l'istanza di `Thread`
  - ma la classe rimane libera di derivare da una qualsiasi altra classe

java.lang.Thread (1)

public class **Thread** extends Object implements Runnable

- A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of `Thread`. This subclass should override the `run` method of class `Thread`. An instance of the subclass can then be allocated and started.

The other way to create a thread is to declare a class that implements the `Runnable` interface. That class then implements the `run` method. An instance of the class can then be allocated, passed as an argument when creating `Thread`, and started

See Also: `Runnable`

java.lang.Thread (2)

Constructor Summary
Thread()   Allocates a new `Thread` object.
Thread(Runnable target)   Allocates a new `Thread` object.

Method Summary	
`void`	**run**() If this thread was constructed using a separate `Runnable` run object, then that `Runnable` object's `run` method is called; otherwise, this method does nothing and returns.
`void`	**start**() Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread.

Creazione di Thread in Java

- La JVM crea un nuovo thread all’invocazione del metodo `start()` di `Thread`
  - Il nuovo thread esegue il metodo `run()` della classe Thread
  - Il thread chiamante ritorna dall’invocazione di `start()` immediatamente

TwoThread.java

```
public class TwoThread extends Thread {

  public void run() {
     for ( int i = 0; i < 10; i++ ) {
        System.out.println("New thread");
     }
  }

  public static void main(String[] args) {
    TwoThread tt = new TwoThread();
    tt.start();

    for ( int i = 0; i < 10; i++ ) {
      System.out.println("Main thread");
    }
  }
}
```

TwoThread.java: Possibili Output

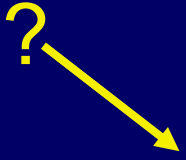
```
Main thread
Main thread
New thread
Main thread
New thread
Main thread
New thread
Main thread
New thread
Main thread
New thread
Main thread
New thread
Main thread
New thread
Main thread
New thread
Main thread
New thread
New thread
```

```
New thread
New thread
New thread
New thread
New thread
New thread
New thread
New thread
New thread
New thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
```

TwoThreadSleep.java

```
public class TwoThreadSleep extends Thread {
  public void run() { loop(); }

  public void loop() {
    // get a reference to the thread running this
    Thread t = Thread.currentThread();
    String name = t.getName();

    System.out.println("just entered loop() - " + name);

    for ( int i = 0; i < 10; i++ ) {
      try { Thread.sleep(200); }
      catch ( InterruptedException x ) { /* ignore */ }

      System.out.println("name=" + name);
    }

    System.out.println("about to leave loop() - " + name);
  }
…
```

TwoThreadSleep.java




```
…
  public static void main(String[] args) {
    TwoThreadSleep tt = new TwoThreadSleep();
    tt.setName("my worker thread");
    tt.start();
    try { Thread.sleep(700); } // pause for a bit
    catch ( InterruptedException x ) { /* ignore */ }
    tt.loop();
  }
}
```

TwoThreadSleep.java: Possibile Output

```
just entered loop() - my worker thread
name=my worker thread
name=my worker thread
name=my worker thread
just entered loop() - main
name=my worker thread
name=main
name=my worker thread
name=main
name=my worker thread
name=main
name=my worker thread
name=main
name=my worker thread
name=main
name=my worker thread
name=main
name=my worker thread
about to leave loop() - my worker thread
name=main
name=main
name=main
name=main
about to leave loop() - main
```

java.lang.Runnable (1)

java.lang Interface Runnable

All Known Implementing Classes: Thread

public abstract interface **Runnable**

The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, `Runnable` is implemented by class `Thread`.

In addition, `Runnable` provides the means for a class to be active while not subclassing `Thread`.

Since: JDK1.0

See Also: `Thread`

java.lang.Runnable (2)

Method Summary	
void	**run**()

Method Detail

run

`public void run()`

When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

The general contract of the method `run` is that it may take any action whatsoever.

See Also: `Thread.run()`

Creazione di Thread Tramite Interfaccia Runnable (1)

```
Class PingPong implements Runnable {
  String word;                    //what word to print
  int delay;                      //how long to pause

  PingPong(String whatToSay, int delayTime) {
    word = whatToSay;
    delay = delayTime;
  }

  // run method…
  // main method…
}
```

Creazione di Thread
Tramite Interfaccia Runnable (2)

```
public void run() {
  try {
    for(;;) {
      System.out.print(word + “ “);
      Thread.sleep(delay); //wait until next time
    }
  } catch (InterruptedException e) {
    return;       // end this thread
  }
}
```

Creazione di Thread Tramite Interfaccia Runnable (3)

```
public static void main(String[] args) {
  Runnable ping = new PingPong(“ping”, 33);
  Runnable pong = new PingPong(“PONG”,100);
  new Thread(ping).start();
  new Thread(pong).start();
}
```

Possibile Output:

```
ping PONG ping ping PONG ping ping PONG ping ping
PONG ping ping PONG ping ping ping PONG ping ping
PONG ping ping PONG ping ping PONG ping ping PONG
ping ping ping PONG ping ping PONG ping ping PONG
ping ping PONG ping ping PONG ping ping ping PONG
ping ………
```

Interruzione di Thread

- Un thread può interrompere un altro thread invocando il metodo `interrupt()` del corrispondente oggetto `java.lang.Thread`
- In realtà il thread interrompente *segnala* soltanto la richiesta di interruzione cambiando un flag dell'oggetto `Thread`

- Il thread interrotto non *serve* la richiesta di interruzione immediatamente
- Invece solleva una eccezione solo in alcuni *idonei* momenti della sua esecuzione: tipicamente in corrispondenza delle attese (passive)

`java.lang.InterruptedException`

- Il thread interrotto servirà la richiesta di interruzione in opportuni momenti; tra i quali
  - durante l'esecuzione di una `sleep()`
  - al momento dell'invocazione di una `sleep()`
  - durante l'esecuzione di una `wait()`
- Il thread interrotto solleva una `java.lang.InterruptedException`
  - in seguito l'interruzione si considera servita ed il flag di interruzione viene resettato
- Terminazione in stile *cooperativo*

Istigazione al Suicidio dei Thread

- In questo modo si possono programmare thread che siano interrotti solo nei punti voluti, senza “traumi”
- N.B. le prime implementazioni della JVM hanno mostrato tutti i problemi delle soluzioni precedenti basate su interruzioni “brutali”
  - Vedi metodi `Thread.stop()/suspend()/resume()`
- Queste hanno senso per “unità di allocazione delle risorse”, come i processi
- Ma non per i thread, che condividendo la spazio di indirizzamento, finiscono per essere intrensicamente accoppiati tra loro e non possono essere terminati “separatamente” l’uno dall’altro

SleepInterrupt.java (1)

```
public class SleepInterrupt implements Runnable {

  public void run() {
    try {
      System.out.println(
        "in run() - about to sleep for 20 seconds");
      Thread.sleep(20000);
      System.out.println("in run() - woke up");
    } catch ( InterruptedException x ) {
      System.out.println(
        "in run() - interrupted while sleeping");
      return;
    }

    System.out.println("in run() - doing stuff after nap");
    System.out.println("in run() - leaving normally");
  }…
```

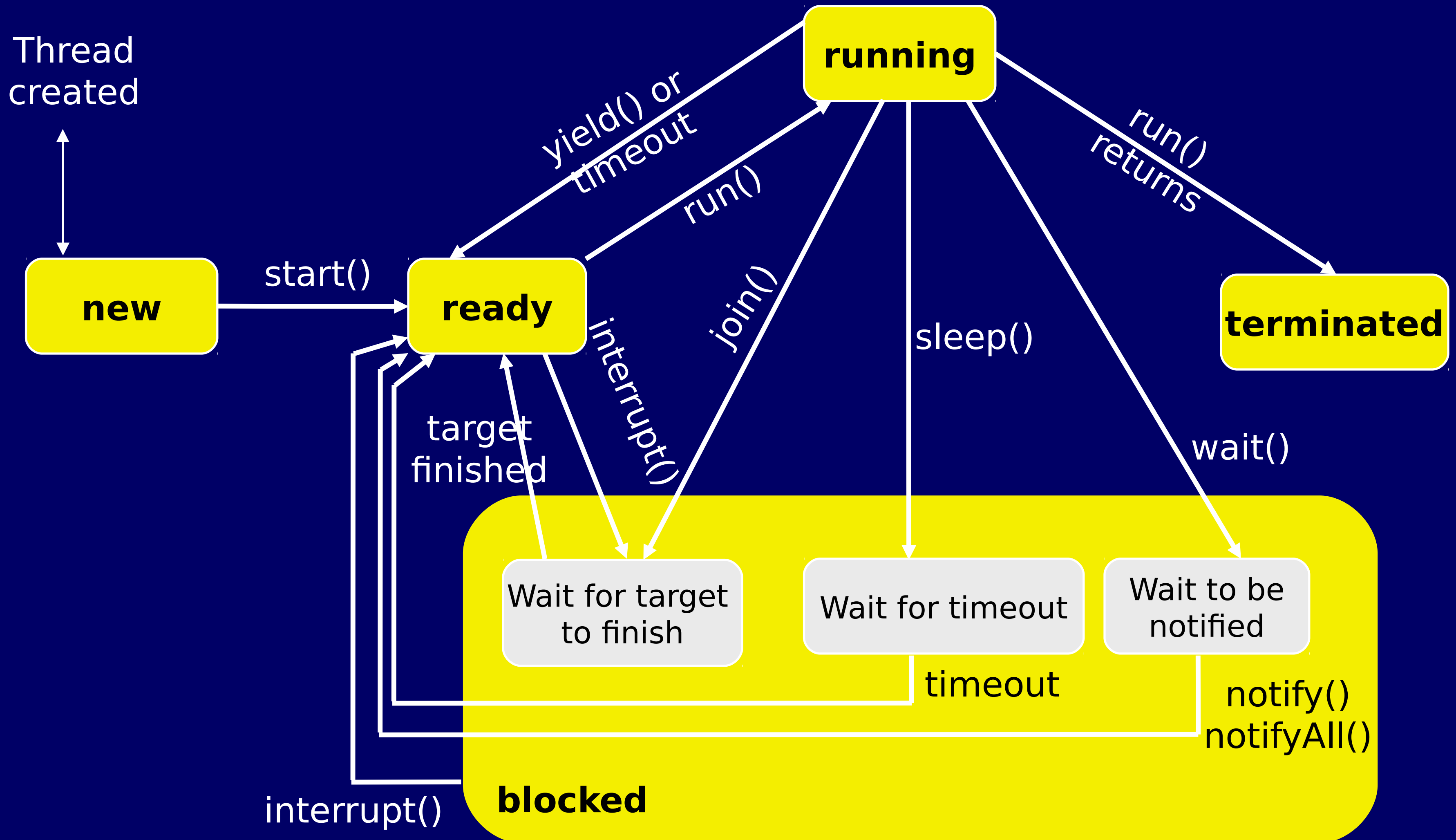
SleepInterrupt.java (2)

```
…
  public static void main(String[] args) {
      SleepInterrupt si = new SleepInterrupt();
      Thread t = new Thread(si);
      t.start();

      // Be sure that the new thread gets a chance to
      // run for a while.
      try { Thread.sleep(2000); }
      catch ( InterruptedException x ) { }

      System.out.println(
         "in main() - interrupting other thread");
      t.interrupt();
      System.out.println("in main() - leaving");
  }
}
```

PendingInterrupt.java

```
public class PendingInterrupt extends Object {
  public static void main(String[] args) {
    if ( args.length > 0 ) {
        Thread.currentThread().interrupt();
    }
    long startTime = System.currentTimeMillis();
    try {
      Thread.sleep(2000);
      System.out.println("was NOT interrupted");
    } catch ( InterruptedException x ) {
      System.out.println("was interrupted");
    }
    System.out.println(
      "elapsedTime=" + (System.currentTimeMillis() - startTime ));
  }
}
```

Diagramma degli Stati/Stati Interrompibili



join() di un Thread

java.lang
Class Thread

java.lang.Object
└ **java.lang.Thread**

All Implemented Interfaces: Runnable

Method Summary

void	`join()` Waits for this thread to die.
void	`join(long millis)` Waits at most `millis` milliseconds for this thread to die.
void	`join(long millis, int nanos)` Waits at most `millis` milliseconds plus `nanos` nanoseconds for this thread to die.

Pericolo di Interferenza

- Diversi thread possono condividere degli oggetti a cui accedono concorrentemente
- E' necessario sincronizzare gli accessi agli oggetti condivisi
- In caso contrario si possono verificare interferenze ed errori dipendenti dalla particolare s.e.a. o seq. di interleaving adottata dalla JVM

CorruptWrite.java (1)

```
public class CorruptWrite extends Object {
  private String fname;
  private String lname;

  public void setNames(String firstName, String lastName) {
    print("entering setNames()");
    fname = firstName;
    // A thread might be swapped out here…
    if ( fname.length() < 5 ) {
      try {Thread.sleep(1000);} catch(InterruptedException x){ }
    } else {
      try {Thread.sleep(2000);} catch(InterruptedException x){ }
    }
    lname = lastName;
    print("leaving setNames() - " + lname + ", " + fname);
  }
…
```

CorruptWrite.java (2)

```
…
  public static void print(String msg) {
    String threadName = Thread.currentThread().getName();
    System.out.println(threadName + ": " + msg);
  }

  public static void main(String[] args) {
    final CorruptWrite cw = new CorruptWrite();
    Runnable runA = new Runnable() {
      public void run() { cw.setNames("George","Washington");}
    };
    Thread threadA = new Thread(runA, "threadA");
    threadA.start();

    try { Thread.sleep(200); } catch (InterruptedException x) { }
    Runnable runB = new Runnable() {
       public void run() { cw.setNames("Abe","Lincoln");     }
    };
    Thread threadB = new Thread(runB, "threadB");
    threadB.start();
  }
}
```

CorruptWrite.java: Output

```
$ java CorruptWrite
threadA: entering setNames()
threadB: entering setNames()
threadB: leaving setNames() - Lincoln, Abe
threadA: leaving setNames() - Washington, Abe
```

Competizione ed Objects Locking

- La JVM associa un semaforo binario in stile competitivo (mutex ricorsivo) ad ogni oggetto
- Utilizzando la parola chiave `synchronized`, un blocco di codice può entrare in sezione critica "bloccando" un qualsiasi oggetto:

```
synchronized (object) {        ← lock
    // Sezione Critica sull'oggetto object
}                              ← unlock
```

- anche interi metodi possono essere eseguiti in sezione critica >>

Method Objects Locking

```
modifier synchronized type method_name() {
   …
}
```

```
modifier type method_name() {
   synchronized (this) {
      …
   }
}
```

FixedWrite.java

```
public class CorruptWrite extends Object {
  private String fname;
  private String lname;

  public synchronized void setNames(String firstName,
                                      String lastName) {
    print("entering setNames()");
    fname = firstName;
    // A thread might be swapped out here…
    if ( fname.length() < 5 ) {
      try {Thread.sleep(1000);} catch(InterruptedException x){ }
    } else {
      try {Thread.sleep(2000);} catch(InterruptedException x){ }
    }
    lname = lastName;
    print("leaving setNames() - " + lname + ", " + fname);
  }
…
```

FixedWrite.java: Output

```
$ java FixedWrite
threadA: entering setNames()
threadA: leaving setNames() - Washington, George
threadB: entering setNames()
threadB: leaving setNames() - Lincoln, Abe
```

Java Monitor vs Monitor (1)

- In Java un qualsiasi oggetto è una particolare implementazione del concetto di *monitor*: un oggetto java è associato ad un *mutex ricorsivo* tramite il quale è possibile disciplinare gli accessi da parte di diversi thread allo stesso oggetto
  - *N.B.*: il compilatore ottimizza fortemente questi aspetti al punto che gli oggetti che non ne fanno uso finiscono per non avere praticamente overhead
- In Java, un oggetto è anche associato ad una *variabile condizione* da intendersi a sua volta associato al monitor
- Ed è forse questo l’aspetto più controverso dei *Java Monitor* (ovvero l’implementazione del concetto di Monitor della piattaforma Java)

Java Monitor vs Monitor (2)

- Mentre risulta naturale avere un mutex associato ad ogni oggetto per realizzare la semantica mutuamente esclusiva degli accessi allo stesso da parte dei thread, la scelta di associare una *sola* variabile condizione risulta ampiamente discutibile
  - In molti problemi si usano *diverse* variabili condizioni all’interno dello stesso monitor
  - Basti solo pensare al più classico dei problemi: il produttore consumatori
    - Var. condizione: *NON_PIENO* & *NON_VUOTO*

Java Monitor vs Monitor (3)

- Rispetto a quanto presentato astrattamente, i *Java Monitor* presentano anche altre differenze:
  - le `notify()` e le `notifyAll()`, possono essere eseguite in qualsiasi punto e non solo alla fine dei metodi
  - se un thread possiede il monitor di un oggetto può chiamare direttamente od indirettamente i metodi `synchronized` della stessa istanza senza pericolo di stallo
  - non tutti i metodi vengono eseguiti in sezione critica
  - è possibile acquisire il monitor di un oggetto anche al di fuori dei metodi della classe di cui è istanza
- Quindi, quale semantica adottano:

  signal_continue o *signal_wait*?

Thread in Java: Cooperazione

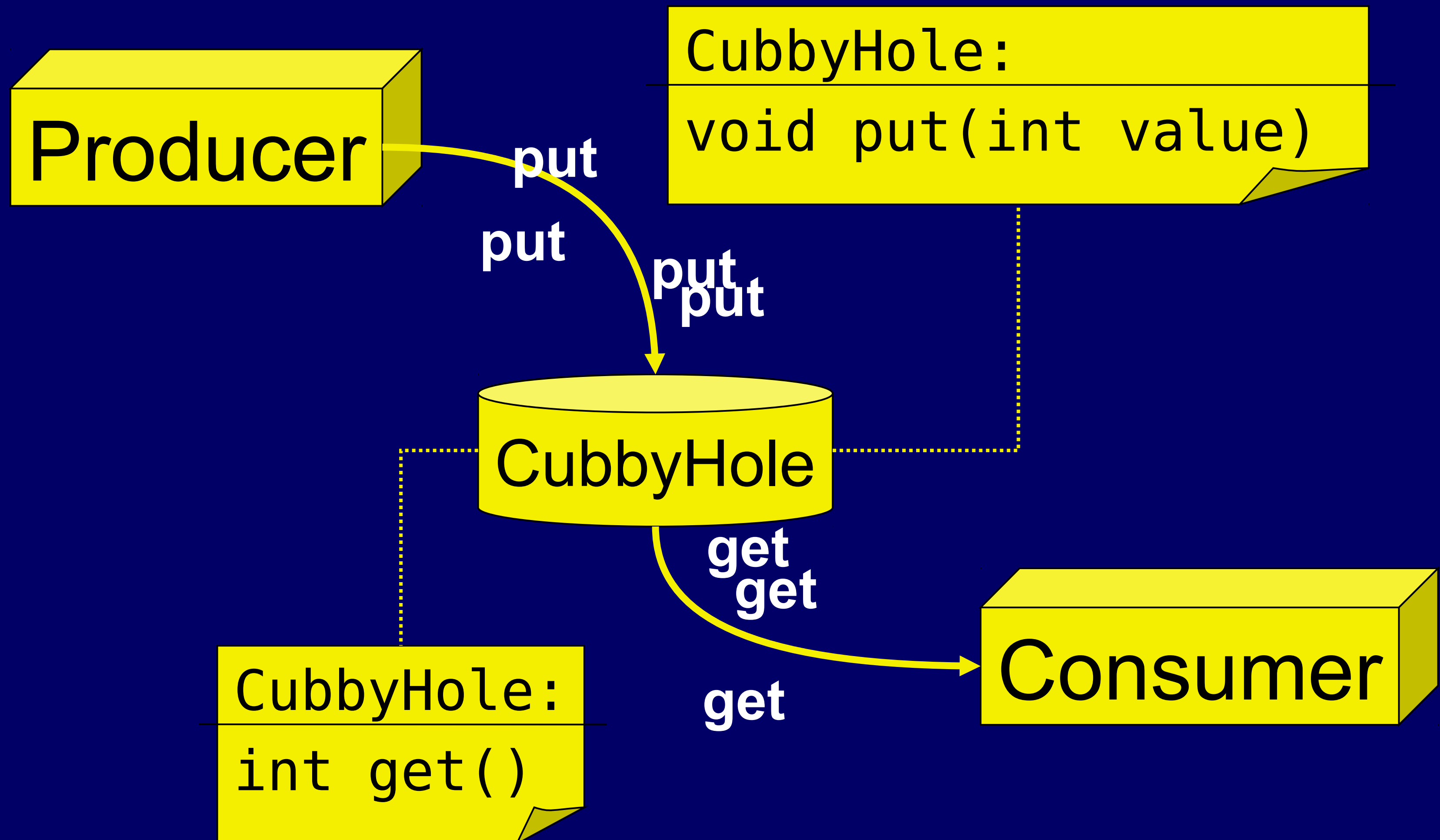
- La JVM associa anche una coda di attesa ad ogni oggetto. Gli oggetti possono *anche* essere visti alla stregua di variabili condizioni
- I thread che sono interessati alla condizione associata ad un oggetto `object` eseguono

  `object.wait()`

  e vengono sospesi in attesa passiva dopo aver rilasciato il monitor
- Altri thread possono segnalare la condizione:

  `object.notify()` oppure `object.notifyAll()`

Produttori/Consumatori / Buffer Unitario



ProducerConsumerTest.java

```
public class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
     …
}
```

Classe Producer

```
public class Producer extends Thread {
  private CubbyHole cubbyhole;
  private int number;

  public Producer(CubbyHole c, int number) {
    cubbyhole = c;
    this.number = number;
  }

  public void run() {
    for (int i = 0; i < 10; i++) {
       cubbyhole.put(i);
       System.out.println("Producer #" +
                          this.number + " put: " + i);
       try {
         sleep((int)(Math.random() * 100));
       } catch (InterruptedException e) { }
    }
  }
}
```

Classe Consumer

```
public class Consumer extends Thread {
  private CubbyHole cubbyhole;
  private int number;

  public Consumer(CubbyHole c, int number) {
    cubbyhole = c;
    this.number = number;
  }

  public void run() {
    int value = 0;
    for (int i = 0; i < 10; i++) {
       value = cubbyhole.get();
       System.out.println("Consumer #" + this.number
                                  +" got: " + value);
    }
  }
}
```

Sincronizzazioni Necessarie

- Due tipologie di problemi di sincronizzazione:
  - gli accessi a `CubbyHole` devono essere *sincronizzati* di modo che *Producer* e *Consumer* non effettuino scritture o letture multiple
  - gli accessi devono essere coordinati di modo che
    - *Consumer* consumi solo quando esiste qualcosa da consumare
    - *Producer* produca solo quando esiste lo spazio per accogliere il valore prodotto

Objects Locking: Producer/Consumer

```
public class CubbyHole {
  private int contents;
  private boolean available = false;

  public synchronized int get()
  {
      ...
  }

  public synchronized void put(int value)
  {
      ...
  }
}
```

Istanza di CubbyHole Locked (dal Consumer)

Istanza di CubbyHole Unlocked (dal Consumer)

Istanza di CubbyHole Locked (dal Producer)

Istanza di CubbyHole Unlocked (dal Producer)

Fenomeni di Interferenza!

- Consumer troppo lento:
  - …
  - `Consumer #1 got: 3`
  - `Producer #1 put: 4`
  - `Producer #1 put: 5`
  - `Consumer #1 got: 5`
  - …
- Consumer troppo veloce:
  - …
  - `Producer #1 put: 4`
  - `Consumer #1 got: 4`
  - `Consumer #1 got: 4`
  - `Producer #1 put: 5`
  - …

Le Primitive di `wait()` & `signal()` di Hoare in Java

- Java supporta la cooperazione tramite primitive *wait()* e *signal()* nello stile delle variabili condizione di C.A.R. Hoare
  - `wait()` e `notify()` in java
- Le primitive in questione sono metodi dalla classe `Object`
- I thread possono sincronizzarsi sul verificarsi di certe condizioni/eventi

java.lang

Class Object

java.lang.Object

Method Summary	
void	**notify**() Wakes up a single thread that is waiting on this object's monitor.
void	**notifyAll**() Wakes up all threads that are waiting on this object's monitor.
void	**wait**() Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	**wait**(long timeout) Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	**wait**(long timeout, int nanos)

wait()/notify() (1)

```
public synchronized int get() {
  While (!available) {
    try {
      // wait for Producer to put value
      wait();
    } catch (InterruptedException e) {}
  }
  available = false;
  // notify Producer that value has been retrieved
  notifyAll();
  return contents;
}
```

La wait() va sempre utilizzata in un ciclo che verifica la condizione a cui è associata

Operazione atomica:
- *Lock su CubbyHole rilasciato*
- *Thread sospeso*

wait()/notify() (2)

```
public synchronized void put(int value) {
  while (available) {
    try {
      // wait for Consumer to get value
      wait();
    } catch (InterruptedException e) {}
  }
  contents = value;
  available = true;
  // notify Consumer that value has been set
  notifyAll();
}
```

Operazione atomica:

- *Lock su CubbyHole rilasciato*
- *Thread sospeso*

Considerazioni Finali sui Java Monitor

```
public class IMSE {

    public static void main(String args[])
                        throws InterruptedException {
        Object o = new Object();
        o.wait();
    }
}
    /* $java IMSE
        Exception in thread "main"
                    java.lang.IllegalMonitorStateException
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Object.java:502)
        at ISME.main(ISME.java:5)
     */
```

```
#include <pthread.h>
int pthread_cond_wait(    pthread_cond_t *cond,
                          pthread_mutex_t *mutex);
```

- `cond` è una variabile condizione
- `mutex` è un mutex associato alla variabile

✔ al momento della chiamata il mutex <u>deve</u> essere bloccato

Esercizi

Esercizio: scrivere un programma java per risolvere il problema molti produttori, molti consumatori, con un buffer circolare di comunicazione capace di ospitare N messaggi

Esercizio: scrivere un programma java per risolvere il problema dei cinque filosofi mangiatori

Thread Specific Data

- Dati privati ad un thread possono essere ottenuti utilizzando `java.lang.ThreadLocal`
- Si tratta essenzialmente di una ***Map*** associata all’oggetto istanza di `java.lang.Thread`
- I valori che restituisce dipendono dal thread chiamante
- `java.lang.InheritableThreadLocal` è una versione di `ThreadLocal` che viene automaticamente ereditata dal thread genitore

java.lang.ThreadLocal (1)

java.lang

Class ThreadLocal

java.lang.Object
  └**java.lang.ThreadLocal**

Direct Known Subclasses:

InheritableThreadLocal

---

public class **ThreadLocal**
extends Object

This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its `get` or `set` method) has its own, independently initialized copy of the variable. `ThreadLocal` instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).

Each thread holds an implicit reference to its copy of a thread-local variable as long as the thread is alive and the `ThreadLocal` instance is accessible; after a thread goes away, all of its copies of thread-local instances are subject to garbage collection (unless other references to these copies exist).

Since: 1.2

java.lang.ThreadLocal (2)

Constructor Summary
ThreadLocal()

Method Summary	
Object	get() Returns the value in the current thread's copy of this thread-local variable.
protected Object	initialValue() Returns the current thread's initial value for this thread-local variable.
void	set(Object value) Sets the current thread's copy of this thread-local variable to the specified value.

ThreadId.java

```
public class ThreadID extends ThreadLocal {
   private int nextID;

   public ThreadID() { nextID = 10001;  }

   private synchronized Integer getNewID() {
        Integer id = new Integer(nextID++);
        return id;
   }

   // override ThreadLocal's version
   protected Object initialValue() {
        print("in initialValue()");
        return getNewID();
   }

   public int getThreadID() {
        Integer id = (Integer) get();
        return id.intValue(); //get() the calling thread's ID
   }
```

ThreadIDMain.java (1)

```
public class ThreadIDMain implements Runnable {
  private ThreadID var;

  public ThreadIDMain(ThreadID var) {
    this.var = var;
  }

  public void run() {
    try {
       print("var.getThreadID()=" + var.getThreadID());
       Thread.sleep(2000);
       print("var.getThreadID()=" + var.getThreadID());
      } catch ( InterruptedException x ) {
           // ignore
      }
  }
  …
}
```

ThreadIDMain.java (2)

```
public class ThreadIDMain implements Runnable {
   …
   private static void print(String msg) {
        String name = Thread.currentThread().getName();
        System.out.println(name + ": " + msg);
   }
   public static void main(String[] args) {
      ThreadID tid = new ThreadID();
      ThreadIDMain shared = new ThreadIDMain(tid);
      try {
        Thread threadA = new Thread(shared, "threadA");
        threadA.start();
        Thread.sleep(500);
        Thread threadB = new Thread(shared, "threadB");
        threadB.start();
        Thread.sleep(500);
        Thread threadC = new Thread(shared, "threadC");
        threadC.start();
      } catch ( InterruptedException x ) { }
   }
}
```

ThreadIDMain.java: Possibile Output

```
$ java ThreadIDMain
threadA: in initialValue()
threadA: var.getThreadID()=10001
threadB: in initialValue()
threadB: var.getThreadID()=10002
threadC: in initialValue()
threadC: var.getThreadID()=10003
threadA: var.getThreadID()=10001
threadB: var.getThreadID()=10002
threadC: var.getThreadID()=10003
```

java.lang.InheritableThreadLocal (1)

java.lang
Class InheritableThreadLocal

```
java.lang.Object
  └java.lang.ThreadLocal
      └java.lang.InheritableThreadLocal
```

public class **InheritableThreadLocal**
extends ThreadLocal

This class extends `ThreadLocal` to provide inheritance of values from parent thread to child thread: when a child thread is created, the child receives initial values for all inheritable thread-local variables for which the parent has values. Normally the child's values will be identical to the parent's; however, the child's value can be made an arbitrary function of the parent's by overriding the `childValue` method in this class.

Inheritable thread-local variables are used in preference to ordinary thread-local variables when the per-thread-attribute being maintained in the variable (e.g., User ID, Transaction ID) must be automatically transmitted to any child threads that are created.

Since: 1.2

java.lang.InheritableThreadLocal (2)

Constructor Summary
InheritableThreadLocal()

Method Summary	
protected Object	**childValue**(Object parentValue) Computes the child's initial value for this inheritable thread-local variable as a function of the parent's value at the time the child thread is created.

Methods inherited from class java.lang.ThreadLocal
get, initialValue, set

Method Detail

childValue

`protected Object childValue(Object parentValue)`

Computes the child's initial value for this inheritable thread-local variable as a function of the parent's value at the time the child thread is created. This method is called from within the parent thread before the child is started.

This method merely returns its input argument, and should be overridden if a different behavior is desired.

Parameters: `parentValue` - the parent thread's value

Returns: the child thread's initial value

Esercizi

Esercizio: riscrivere il programma java per risolvere il problema molti produttori, molti consumatori, con un buffer circolare di comunicazione capace di ospitare N messaggi in modo da disporre di un file di log per ogni thread

Esercizio: implementare l'algoritmo di ordinamento merge-sort in versione parallela multi-thread.