Università degli Studi dell'Insubria
Dipartimento di Scienze Teoriche e Applicate

Programmazione Concorrente e Distribuita

Esempi classici di sincronizzazione tra task

Luigi Lavazza
Dipartimento di Scienze Teoriche e Applicate
luigi.lavazza@uninsubria.it

PRODUTTORE CONSUMATORE

Produttore-Consumatore: il problema

- Due task (il produttore e il consumatore) comunicano attraverso un buffer.
  - Il buffer ha capacità finita (ovviamente).
- Compito del produttore è generare dati e depositarli nel buffer.
- Il consumatore utilizza i dati prodotti, rimuovendoli di volta in volta dal buffer.
- I due task producono e consumano continuamente, ma con frequenza variabile e non nota a priori.

- Problemi:
  - Gestire l'accesso concorrente al buffer (sezione critica)
    - Questo sappiamo farlo
  - Gestire il comportamento del produttore quando il buffer è pieno.
  - Gestire il comportamento del consumatore quando il buffer è vuoto.

Produttore-Consumatore: la soluzione

- Il produttore deve sospendere la propria esecuzione se il buffer è pieno
- Il consumatore si sospende se il buffer è vuoto

- Quando il consumatore preleva un elemento dal buffer pieno “sveglia” il produttore, che ricomincerà quindi a depositare elementi nel buffer
- Quando il produttore deposita un elemento nel buffer vuoto “sveglia” il consumatore, che ricomincerà quindi a prelevare elementi dal buffer

Produttore-Consumatore: la soluzione

- La soluzione prospettata può essere implementata tramite le primitive di comunicazione tra Thread:
  - **`synchronized`**
  - **`wait()`**
  - **`notify()`**
  - **`notifyAll()`.**
- Attenzione:
  - una soluzione errata potrebbe dar luogo ad una race condition e/o ad un deadlock.

Esempio di implementazione errata

```
public class CellaCondivisa {
  int valore ;
  public int getValore() {
    return valore;
  }
  public void setValore(int valore) {
    this.valore = valore;
  }
}
```

Iniziamo con un buffer di capienza unitaria.
Produttore e consumatore dovrebbero alternarsi.
Dovrebbe iniziare il produttore

Esempio di implementazione errata

```
import java.util.concurrent.ThreadLocalRandom;
class Produttore extends Thread{
  CellaCondivisa cella;
  public Produttore(CellaCondivisa c){
    this.cella=c;
  }
  public void run(){
    for(int i=1; i<=10; ++i){
      try {
       Thread.sleep(ThreadLocalRandom.current().nextInt(10,100));
      } catch (InterruptedException e) {  }
      synchronized(cella){
        int v=(int)(100*Math.random());
        cella.setValore(v);
        System.out.print(" P"+i+"("+v+")");
      }  // end synchronized
    }
  }
}
```

Sezione critica: non ci sono race condition sul valore

Esempio di implementazione errata

```
import java.util.concurrent.ThreadLocalRandom;
class Consumatore extends Thread{
  CellaCondivisa cella;
  public Consumatore(CellaCondivisa c){
    this.cella=c;
  }
  public void run(){
    int v;
    for(int i=1; i<=10; ++i){
      synchronized(cella){
        v=cella.getValore();
        System.out.print(" C"+i+"("+v+")");
      }  // end synchronized
      try {
        Thread.sleep(ThreadLocalRandom.current().
                     nextInt(10,100));
      } catch (InterruptedException e) {  }
    }
  }
}
```

Sezione critica: non ci sono race condition sul valore

Esempio di implementazione errata

```
public class ProdCons{
  public static void main(String[] args){
    CellaCondivisa cella=new CellaCondivisa();
    new Produttore(cella).start();
    new Consumatore(cella).start();
  }
}
```

- Un possibile output:

  C1(0) C2(0) P1(73) P2(90) C3(90) C4(90) C5(90) P3(84) C6(84) P4(96) P5(74) C7(74) C8(74) C9(74) P6(33) C10(33) P7(74) P8(19) P9(92) P10(21)

- Chiaramente errato:
  - Inizia il consumatore
  - Diversi elementi sono sovrascritti (quindi persi).
  - Diversi elementi sono letti più volte

Versione thread-safe della cella (implementazione errata)

```
public class CellaCondivisa {
  int valore ;

  public synchronized int getValore() {
    System.out.print("Viene letto "+valore);
    return valore;
  }

  public synchronized void setValore(int valore) {
    System.out.print("Viene scritto "+valore);
    this.valore = valore;
  }
}
```

Esempio di implementazione corretta

- Supponiamo di avere a disposizione una classe che implementa CellaCondivisa correttamente
  - In modo che i task produttore e consumatore non debbano preoccuparsi della sincronizzazione
  - Cioè, vengono messi in attesa quando opportuno, senza che se ne accorgano
- Vediamo il codice dei task e il main.

Esempio di implementazione corretta

```
import java.util.concurrent.ThreadLocalRandom;

class Produttore extends Thread{
  CellaCondivisa cella;
  public Produttore(CellaCondivisa c){
    this.cella=c;
  }
  public void run(){
    for(int i=1; i<=10; ++i){
      try {
        Thread.sleep(ThreadLocalRandom.current().
                     nextInt(10, 100));
      } catch (InterruptedException e) {  }
      int v=(int)(100*Math.random());
      cella.setValore(v);
    }
  }
}
```

Più semplice di così non si può …

Esempio di implementazione corretta

```
import java.util.concurrent.ThreadLocalRandom;

class Consumatore extends Thread{
  CellaCondivisa cella;
  public Consumatore(CellaCondivisa c){
    this.cella=c;
  }
  public void run(){
    int v;
    for(int i=1; i<=10; ++i){
      v=cella.getValore();
    }
    try {
      Thread.sleep(ThreadLocalRandom.current().
                   nextInt(10, 100));
    } catch (InterruptedException e) {  }
  }
}
```

Esempio di implementazione corretta

```
public class ProdCons{
  public static void main(String[] args){
    CellaCondivisa cella=new CellaCondivisa();
    new Produttore(cella).start();
    new Consumatore(cella).start();
  }
}
```

Esempio di implementazione corretta

- Scriviamo la classe CellaCondivisa in modo che produzione e consumo procedano come desiderato.
  - Oltre a garantire accesso esclusivo alla cella.

```
public class CellaCondivisa {
  static final int BUFFERSIZE = 1;
  private int numItems = 0;
  private int valore;

  public int getCurrentSize(){
    return numItems;
  }
```

Esempio di implementazione corretta

```
public synchronized int getValore() {
    if(numItems==0){
      try {
        wait();
      } catch (InterruptedException e) {  }
    }
    numItems--;
    System.out.println("Letto "+valore);
    notify();
    return valore;
 }
```

Chi cerca di leggere viene fermato se non c'è niente da leggere.

A questo punto il buffer non è più pieno: svegliamo un eventuale produttore in attesa.

Esempio di implementazione corretta

```
public synchronized void setValore(int v) {
    if(numItems==BUFFERSIZE){
      try {
        wait();
      }
       catch (InterruptedException e) {  }
    }
    valore=v;
    System.out.println("Scritto "+valore);
    numItems++;
    notify();
  }
}
```

Chi sta cercando scrivere e trova il buffer pieno di ferma in attesa che non sia più pieno

A questo punto il buffer non è sicuramente vuoto: si può svegliare un eventuale consumatore in attesa.

Esempio corretto: output

- Possibile output corretto:

```
Scritto 96
Letto 96
Scritto 13
Letto 13
Scritto 82
Letto 82
Scritto 57
Letto 57
Scritto 49
Letto 49
Scritto 14
Letto 14
Scritto 21
Letto 21
Scritto 55
Letto 55
Scritto 56
Letto 56
Scritto 22
Letto 22
```

Codice “parlante”

```
public synchronized int getValore() {
   if(numItems==0){
     try {
       wait();
     } catch (InterruptedException e) {  }
   }
   numItems--;
   System.out.println("Letto "+valore);
   notify();
   return valore;
 }
 public synchronized void setValore(int v) {
   if(numItems==BUFFERSIZE){
     try {
       wait();
     } catch (InterruptedException e) {  }
   }
   valore=v;
   System.out.println("Scritto "+valore);
   numItems++;
   notify();
 }
```

Facciamo println in sezione critica! Quindi la lettura e la println stanno insieme!

Facciamo println in sezione critica! Quindi la scrittura e la println stanno insieme!

Esempio corretto: output

- Il posizionamento delle println nelle sezioni syncronized garantisce l'ordinamento delle println coerente con l'esecuzione dei thread.

```
Scritto 87
Letto 87
Scritto 86
Letto 86
Scritto 64
Letto 64
Scritto 2
Letto 2
Scritto 93
Letto 93
Scritto 78
Letto 78
Scritto 87
Letto 87
Scritto 37
Letto 37
Scritto 24
Letto 24
Scritto 32
Letto 32
```

ATTENZIONE

- È importante stare attenti a dove si fanno **`notify`** e **`wait`**.

```
public synchronized void setItem(int v)
    throws InterruptedException {
    if(numItems==BUFFERSIZE){
      wait();
    }
    valore=v;
    numItems++;
    if(numItems==1){
      notify();
    }
  }
```

Appena entrati

Appena prima di uscire

Esempio di implementazione errata

```
void outState(String s) {
   System.out.println(Thread.currentThread().getName()+
                      s);
 }
 public synchronized int getItem()
                          throws InterruptedException {
   outState(" entered get ");
   if(numItems==0){
     outState(" going to wait on get ");
     wait();
     outState(" going to notify in get ");
     notify();
   }
   numItems--;
   outState(" exiting get");
   return valore;
 }
```

KO

Esempio di implementazione errata

```
public synchronized void setItem(int v)
                                    throws InterruptedException {
  outState(" entered set ");
  if(numItems==BUFFERSIZE){
    outState(" going to wait on set ");
    wait();
    outState(" going to notify in set ");
    notify();
  }
  valore=v;
  numItems++;
  outState(" exiting set ");
}
```

KO

Esempio di implementazione errata

- Esempio di output scorretto:
  - Prod entered set
  - Prod exiting set
  - Prod entered set
  - Prod going to wait on set
  - Cons entered get
  - Cons exiting get
  - Cons entered get
  - Cons going to wait on get
  - [deadlock]
- Perché si blocca?
- Per capirlo, sfruttiamo i metodi «parlanti» della cella

Esempio di implementazione errata

- Esempio di output scorretto:
  - Prod entered set
  - Prod exiting set
  - Prod entered set
  - Prod going to wait on set
  - Cons entered get
  - Cons exiting get
  - Cons entered get
  - Cons going to wait on get
  - [deadlock]

Il produttore riempie il buffer

Il produttore trova il buffer pieno e si sospende

Il consumatore svuota il buffer

Il consumatore trova il buffer vuoto e si sospende

- Perché si blocca?
- Per capirlo, sfruttiamo i metodi «parlanti» della cella

Usiamo un buffer più grande

- Finora il buffer **`CellaCondivisa`** conteneva un solo elemento.
- Ingrandiamo il buffer:

Solo agli effetti della sincronizzazione (la cella è sempre unica)

```
public class CellaCondivisa {
  static final int BUFFERSIZE = 4;
  private int numItems = 0;
```

Buffer di 4 elementi

Possibile output

- Possibile output corretto:
  - Scritto 34
  - Letto
  - Scritto 77
  - Scritto 5
  - Scritto 41
  - Scritto 32
  - Letto
  - Letto
  - Letto
  - Letto
  - Scritto 86
  - Letto
  - Scritto 31
  - Letto
  - Scritto 39
  - Letto
  - Scritto 72
  - Letto
  - Scritto 41
  - Letto

Ci sono più scritture e letture consecutivie: OK, il buffer di 4 elementi lo consente. Non ci sono mai overflow o underflow del buffer.

Tanti produttori e tanti consumatori

- Cosa succede se abbiamo
  - Tanti thread che producono
  - Tanti thread che consumano
  - Tutti usando il medesimo buffer
    - Che ha capienza > 1

ProdCons

```
public class ProdCons {
      public static void main(String[] args) {
          CellaCondivisa cella=new CellaCondivisa();
          new Produttore("p1", cella).start();
          new Produttore("p2", cella).start();
          new Consumatore("c1", cella).start();
          new Consumatore("c2", cella).start();
      }
}
```

Produttore

```
public class Produttore extends Thread {
  CellaCondivisa cella;
  public Produttore(String name, CellaCondivisa c){
    super(name);
    this.cella=c;
  }
  public void run(){
    int i=0;
    for(;;){
      cella.setValore(i++);
    }
  }
}
```

Consumatore

```
public class Consumatore extends Thread {
  CellaCondivisa cella;
  int v;
  public Consumatore(String name, CellaCondivisa c){
    super(name);
    this.cella=c;
  }
  public void run(){
    for(;;){
      v=cella.getValore();
    }
  }
}
```

CellaCondivisa

```
public class CellaCondivisa {
  static final int BUFFERSIZE = 4;
  private int valore, numItems = 0;
  void printWithName(String s) {
    System.out.println(Thread.currentThread().getName()+
                       s+valore+"["+numItems+"]");
  }
```

CellaCondivisa

```
public synchronized int getValore() {
  while(numItems==0){
    try { wait(); } catch (InterruptedException e) { }
  }
  numItems--;
  printWithName(" legge ");
  notify();
  return valore;
}
public synchronized void setValore(int v) {
  while(numItems==BUFFERSIZE){
    try { wait(); } catch (InterruptedException e) { }
  }
  valore=v;
  numItems++;
  printWithName(" scrive ");
  notify();
}
}
```

Con tanti produttori e consumatori bisogna usare un ciclo while. Perché?

Con tanti produttori e consumatori bisogna usare un ciclo while. Perché?

Il programma funziona correttamente

```
p1 scrive 34467[1]
p1 scrive 34468[2]
p1 scrive 34469[3]
p1 scrive 34470[4]
c1 legge 34470[3]
c1 legge 34470[2]
p1 scrive 34471[3]
p1 scrive 34472[4]
c1 legge 34472[3]
c1 legge 34472[2]
c1 legge 34472[1]
c1 legge 34472[0]
p2 scrive 34713[1]
p2 scrive 34714[2]
p2 scrive 34715[3]
p2 scrive 34716[4]
c2 legge 34716[3]
c2 legge 34716[2]
c2 legge 34716[1]
c2 legge 34716[0]
```

Il buffer non contiene mai più di 4 elementi o meno di zero.

Cella condivisa scorretta

```
public synchronized int getValore() {
    if(numItems==0){
      try { wait(); } catch (InterruptedException e) { }
    }
    numItems--;
    printWithName(" legge ");
    notify();
    return valore;
  }
  public synchronized void setValore(int v) {
    if(numItems==BUFFERSIZE){
      try { wait(); } catch (InterruptedException e) { }
    }
    valore=v;
    numItems++;
    printWithName(" scrive ");
    notify();
  }
}
```

Il programma NON funziona correttamente

```
c2 legge 58236[32926]
c2 legge 58236[32925]
c2 legge 58236[32924]
c2 legge 58236[32923]
c2 legge 58236[32922]
c2 legge 58236[32921]
c2 legge 58236[32920]
c2 legge 58236[32919]
c2 legge 58236[32918]
c2 legge 58236[32917]
c2 legge 58236[32916]
c2 legge 58236[32915]
c2 legge 58236[32914]
c2 legge 58236[32913]
c2 legge 58236[32912]
c2 legge 58236[32911]
c2 legge 58236[32910]
```

Il buffer ha palesemente subito un overflow, e abbiamo una sequenza esagerata di consumi.

Il problema

```
if(numItems==0){
   wait();
}
numItems--;
```

La lettura e l'aggiornamento di **numItems** dovrebbero stare in una sezione critica.
Il fatto che stiano in un metodo **synchronized** non basta, perché **wait** provoca il rilascio del lock e rende la sezione interrompibile.

Istruzione eseguita	Stato risultante	numItems	Note
if(numItems==0)	exec	0	
wait	blocked	0	
---	ready	1	Un altro thread incrementa **numItems** e fa **notify**
---	ready	0	Un altro thread decrementa **numItems**
numItems--;	exec	-1	ERRORE

La soluzione

```
while(numItems==0){
   wait();
}
numItems--;
```

La scrittura (aggiornamento di **`numItems`**) viene fatta solo in una sezione non interrompibile.
Lettura e scrittura stanno in un metodo **`synchronized`**
`numItemts` viene decrementato solo quando l'istruzione precedente è una verifica **`numItems==0`** che non ha comportato **`wait`** e quindi non ha permesso di interrompere la sezione critica.

Istruzione eseguita	Stato risultante	`numItems`
`while(numItems==0)`	exec	0
wait	blocked	0
---	ready ←	1
---	ready	0
`while(numItems==0)`	exec	0
wait	blocked	0
---	ready ←	1
`while(numItems==0)`	exec	1
`numItems--;`	exec	0

Usiamo `notifyAll`

```
public synchronized int getValore() {
  while(numItems==0){
    try { wait(); } catch (InterruptedException e) { }
  }
  numItems--;
  printWithName(" legge ");
  notifyAll();
  return valore;
}
public synchronized void setValore(int v) {
  while(numItems==BUFFERSIZE){
    try { wait(); } catch (InterruptedException e) { }
  }
  valore=v;
  numItems++;
  printWithName(" scrive ");
  notifyAll();
}
}
```

Il programma funziona correttamente

p2 scrive 88157[1]
p2 scrive 88158[2]
c2 legge 88158[1]
c2 legge 88158[0]
p2 scrive 88159[1]
p2 scrive 88160[2]
p2 scrive 88161[3]
p1 scrive 92010[4]
c1 legge 92010[3]
c1 legge 92010[2]
c1 legge 92010[1]
c1 legge 92010[0]
p1 scrive 92011[1]
p1 scrive 92012[2]

Frammento di output corretto

Qual è l'effetto di `notifyAll`?

- Quando si fa **`notifyAll`**, vengono risvegliati tutti i thread in attesa sulla cella condivisa, sia produttori sia consumatori.
- Poiché tutti i metodi contengono del codice di tipo «**`while(!condizione) {wait}`**», tutti i thread ricominceranno a eseguire dalla valutazione della condizione.
  - Quelli che la trovano soddisfatta proseguono
  - Gli altri tornano a sospendersi con la **`wait`**

Produttore-consumatore con semafori

Produttore-consumatore con semafori

- Abbiamo bisogno di tre semafori
  - Uno per la mutua esclusione
    - Per avere un solo thread nella sezione critica
  - Uno per bloccare il consumatore quando il buffer è vuoto.
  - Uno per bloccare il produttore quando il buffer è pieno.

main

```
import java.util.concurrent.Semaphore;
public class ProdCons {
  public static final Semaphore mutex = new Semaphore(1);
  public static final Semaphore full = new Semaphore(0);
  public static final Semaphore empty =
                        new Semaphore(CellaCondivisa.BUFFERSIZE);
  public static void main(String[] args) {
    CellaCondivisa cella=new CellaCondivisa();
    new Produttore("Prod", cella).start();
    new Consumatore("Cons", cella).start();
  }
}
```

CellaCondivisa

```
public class CellaCondivisa {
  static final int BUFFERSIZE = 1;
  private int numItems = 0;
  private int valore;
  void printWithName(String s) {
    System.out.println(Thread.currentThread().getName()+
                       s+valore+"["+numItems+"]");
  }
  public int getCuttentSize(){
    return numItems;
  }
```

CellaCondivisa

```
public int getValore(){
  int tmp;
  try{
    ProdCons.mutex.acquire();
  } catch(InterruptedException e) {}
  tmp=valore;
  numItems--;
  printWithName(" letto ");
  ProdCons.mutex.release();
  return tmp;
}
public void setValore(int v) {
  try{
    ProdCons.mutex.acquire();
  } catch(InterruptedException e) {}
  valore=v;
  numItems++;
  printWithName(" scritto ");
  ProdCons.mutex.release();
}
}
```

Sezione critica

Sezione critica

Consumatore

```
public class Consumatore extends Thread {
  CellaCondivisa cella;
  int v;
  public Consumatore(String s, CellaCondivisa c){
    super(s);
    this.cella=c;
  }
  public void run(){
    for(;;){
      try{
        ProdCons.full.acquire();
      } catch(InterruptedException e) {}
      v=cella.getValore();
      ProdCons.empty.release();
    }
  }
}
```

Si blocca se il buffer non contiene almeno un elemento

Ora il buffer non è certo più pieno: sblocchiamo eventuali produttori in attesa.

Produttore

```
public class Produttore extends Thread {
  CellaCondivisa cella;
  public Produttore(String s, CellaCondivisa c){
    super(s);
    this.cella=c;
  }
  public void run(){
    int i=0;
    for(;;){
      try{
       ProdCons.empty.acquire();
      } catch(InterruptedException e) {}
      cella.setValore(i++);
      ProdCons.full.release();
    }
  }
}
```

Si blocca se il buffer è pieno

Ora il buffer non è certo più vuoto: sblocchiamo eventuali consumatori in attesa.

Usiamo una coda

- In generale il buffer usato dai produttori e consumatori (possono essere più di uno per tipo) ha dimensioni maggiori di uno.
- In genere gli elementi vengono consumati nello stesso ordine in cui sono prodotti.
- Abbiamo bisogno di una coda (cioè di un buffer FIFO)

Produttore consumatore con semafori

- E coda “artigianale”
  - Thread-safe (nel senso che non ci sono race) ma non bloccante

Coda

```
public class CodaCondivisa {
  static int BUFFERSIZE;
  private int numItems = 0;
  private int[] valori;
  private int first, last; // last is the index of the
                           // most recently inserted item
  public CodaCondivisa(int bufsize) {
    BUFFERSIZE=bufsize;
    first=0; last=0;
    valori=new int[BUFFERSIZE];
  }
  void printWithName(String s, int v) {
    System.out.println(Thread.currentThread().getName()+
                       s+v+"["+numItems+"]");
  }
  public int getCuttentSize(){
    return numItems;
  }
```

Coda

```
public int getItem(){
  int tmp;
  try{
    ProdCons.mutex.acquire();
  } catch(InterruptedException e) {}
  if(numItems==0){
    System.err.print("lettura di buffer vuoto!\n");
    System.exit(0);
  }
  numItems--;
  tmp=valori[first];
  first=(first+1)%BUFFERSIZE;
  printWithName(" letto ", tmp);
  ProdCons.mutex.release();
  return tmp;
}
```

Coda

```
  public void setItem(int v) {
    try{
      ProdCons.mutex.acquire();
    } catch(InterruptedException e) {}
    if(numItems==BUFFERSIZE){
      System.err.print("scrittura di buffer pieno!\n");
      System.exit(0);
    }
    valori[last]=v;
    last=(last+1)%BUFFERSIZE;
    numItems++;
    printWithName(" scritto ", v);
    ProdCons.mutex.release();
  }
}
```

main

```
import java.util.concurrent.Semaphore;
public class ProdCons {
  private static final int bufsize=4;
  public static final Semaphore mutex = new Semaphore(1);
  public static final Semaphore full = new Semaphore(0);
  public static final Semaphore empty = new Semaphore(bufsize);
  public static void main(String[] args) {
    CodaCondivisa cella=new CodaCondivisa(bufsize);
    new Produttore("Prod-1", cella).start();
    new Consumatore("Cons-1", cella).start();
    new Produttore("Prod-2", cella).start();
    new Consumatore("Cons-2", cella).start();
  }
}
```

Produttore

```
public class Produttore extends Thread {
  CodaCondivisa cella;
  public Produttore(String s, CodaCondivisa c){
    super(s);
    this.cella=c;
  }
  public void run(){
    int i=0;
    for(;;){
      try{
        ProdCons.empty.acquire();
      } catch(InterruptedException e) {}
      cella.setItem(i++);
      ProdCons.full.release();
    }
  }
}
```

Consumatore

```
public class Consumatore extends Thread {
  CodaCondivisa cella;
  int v;
  public Consumatore(String s, CodaCondivisa c){
    super(s);
    this.cella=c;
  }
  public void run(){
    for(;;){
      try{
        ProdCons.full.acquire();
      } catch(InterruptedException e) {}
      v=cella.getItem();
      ProdCons.empty.release();
    }
  }
}
```

Esempio di output (frammento)

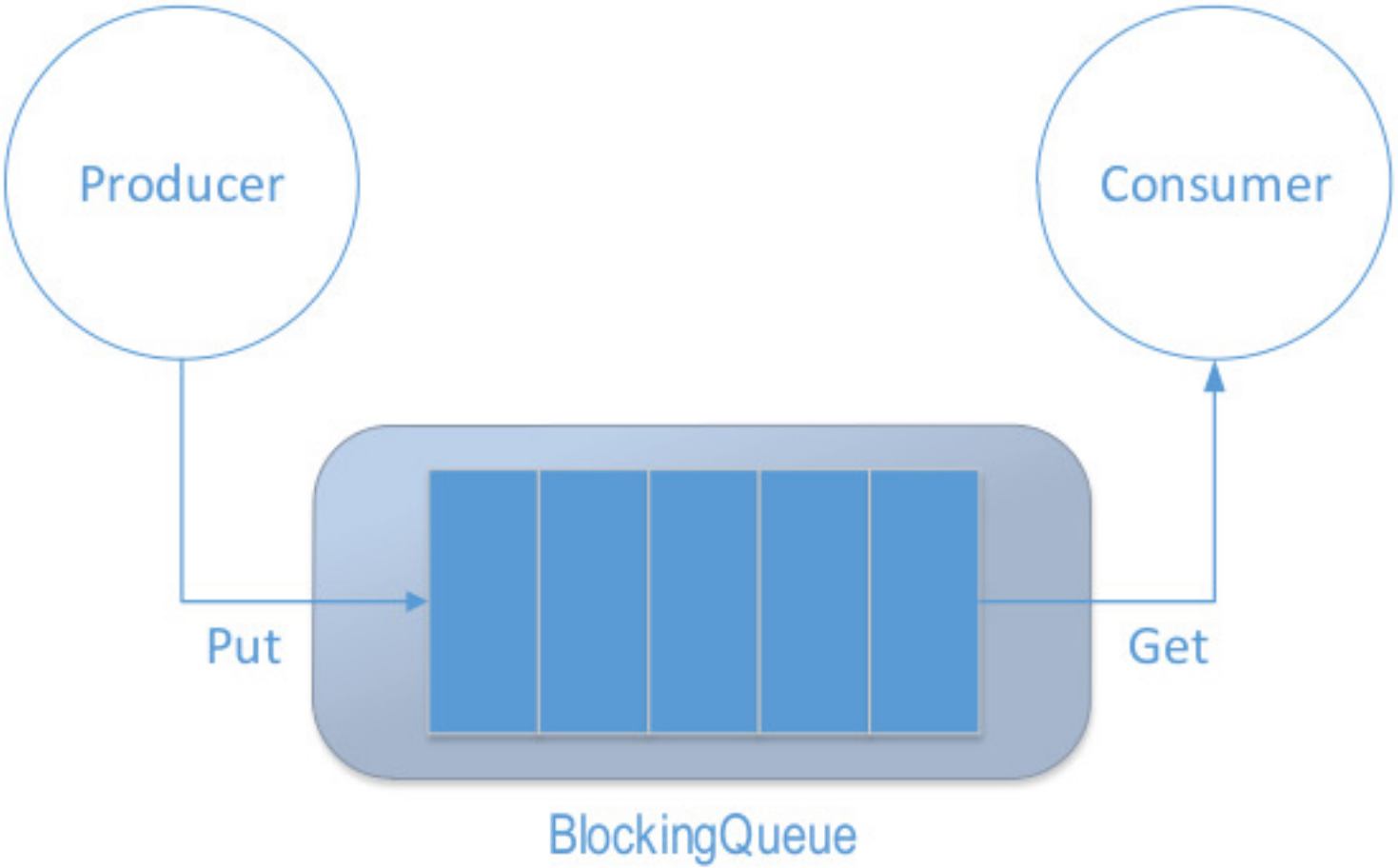
Cons-2 letto 53837[2]
Cons-2 letto 53170[1]
Prod-1 scritto 53171[2]
Cons-1 letto 53838[1]
Prod-2 scritto 53839[2]
Prod-2 scritto 53840[3]
Prod-1 scritto 53172[4]
Cons-2 letto 53171[3]
Cons-1 letto 53839[2]
Cons-2 letto 53840[1]
Prod-2 scritto 53841[2]

I dati sono consumati nello stesso ordine con cui sono stati prodotti

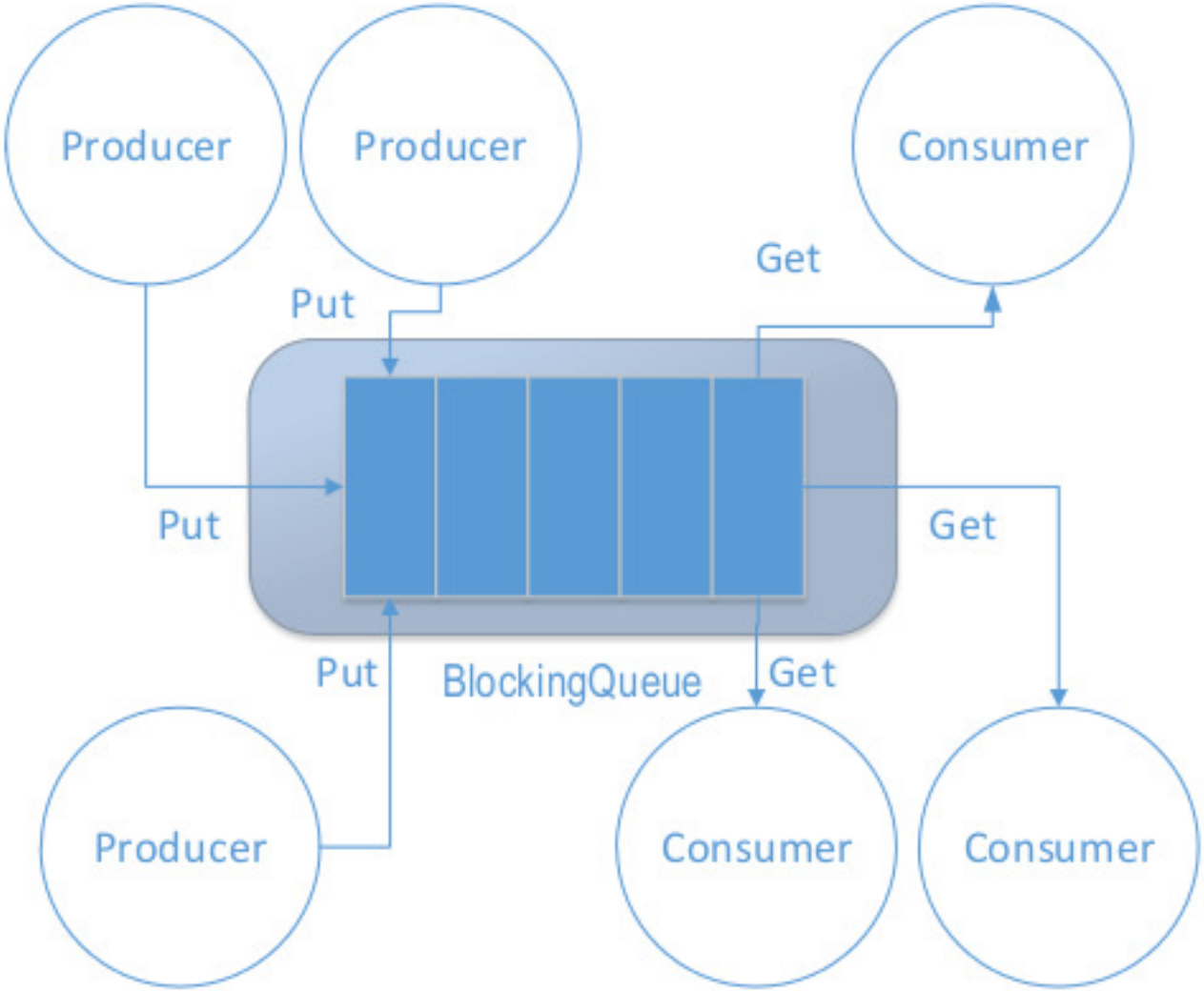
`java.util.concurrent`: BlockingQueue

- **`java.util.concurrent`**: Utility classes commonly useful in concurrent programming
- **`BlockingQueue`**: an interface in the **`java.util.concurrent`** class represents a queue which is thread safe to put into, and take instances from.
- Designed for Producer Consumer model: “FIFO data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.”
- Can be used for multiple producers and multiple consumers.

`java.util.concurrent`: BlockingQueue



`java.util.concurrent`: BlockingQueue



How to use BlockingQueue : Implementations

`BlockingQueue` is an interface, requires its implementations to use it.

- **`ArrayBlockingQueue`**: a bounded, blocking queue that stores the elements internally in an array
- **`LinkedBlockingQueue`**: keeps the elements internally in a linked structure
- **`PriorityBlockingQueue`**: an unbounded concurrent queue of which the elements are ordered according to their natural ordering,
- **`DelayQueue`**: an unbounded concurrent queue keeps the elements internally until a certain delay has expire

Producer-Consumer: BlockingQueue

- **`java.util.concurrent.BlockingQueue`** è una Queue che supporta
  - operazioni di lettura che attendono che il Buffer diventi non-vuoto
  - Operazioni di scrittura che attendono che uno spazio nel Buffer diventi disponibile quando si aggiunge un elemento.
- Definiamo una classe **`Message`** per il contenuto della coda.

```
public class Message {
  private String msg ;
  public Message (String str) {
    this.msg=str;
  }
  public String getMsg() {
    return msg;
  }
}
```

Producer-Consumer con BlockingQueue: main

```
import java.util.concurrent.*;

public class ProdCons {
  static final int queueSize=4;
  public static void main(String[] args) {
    BlockingQueue<Message> queue =
                new ArrayBlockingQueue<Message>(queueSize);
    Producer producer=new Producer(queue);
    Consumer consumer=new Consumer(queue);
    new Thread(producer, "prod-1").start();
    new Thread(consumer, "cons-1").start();
    new Thread(producer, "prod-2").start();
    new Thread(consumer, "cons-2").start();
  }
}
```

Producer-Consumer: BlockingQueue

- Le classi che implementano BlockingQueue sono thread-safe
- In questa implementazione del Producer-Consumer useremo un **`ArrayBlockingQueue`** e i seguenti metodi
  - **`put(E e)`**: usato per inserire elementi nella Queue, se la queue è piena allora il Thread fa **`wait`** in attesa che lo spazio diventi disponibile.
  - **`E take()`**: rimuove e ritorna l'elemento di tipo E dalla testa della Queue, se la queue è vuota il Thread va in **`wait`** in attesa che l'elemento diventi disponibile.
- L'interfaccia BlockingQueue è parte integrante del collections framework di java ed è principalmente utilizzata per la realizzazione del problema del produttore-consumatori.
- Usando le classi che implementano una BlockingQueue non abbiamo bisogno di preoccuparci di attendere che la coda sia non piena o non vuota: è tutto gestito automaticamente dalla coda stessa.

Metodi di una BlockingQueue

- BlockingQueue methods come in four forms:
  1. throws an exception,
  2. returns a special value (either null or false),
  3. blocks the current thread indefinitely until the operation can succeed,
  4. blocks for only a given maximum time limit before giving up.

	Throws exception	Special value	Blocks	Times out
Insert	*add(e)*	*offer(e)*	*put(e)*	*offer(e, time, unit)*
Remove	*remove()*	*poll()*	*take()*	*poll(time, unit)*
Examine	*element()*	*peek()*	-	-

Producer-Consumer con BlockingQueue: produttore

```
import java.util.concurrent.*;
public class Producer implements Runnable {
  private BlockingQueue<Message> queue;
  public Producer(BlockingQueue<Message> q) { this.queue=q }
  public void run(){
    for(int i=0; i<100; i++) {
      Message msg = new Message ("dato_" + i);
      try {
        Thread.sleep(10);
        queue.put(msg);   // produce messages
        System.out.println(Thread.currentThread().getName()+
                           " produced " + msg.getMsg());
      } catch (InterruptedException e) { }
    }
    try { queue.put(new Message ("exit"));
    } catch (InterruptedException e) { }
    System.out.println("Producer finished");
  }
}
```

Producer-Consumer con BlockingQueue: consumatore

```
import java.util.concurrent.*;
public class Consumer implements Runnable {
  private BlockingQueue<Message> queue;
  public Consumer (BlockingQueue<Message> q) {
    this.queue=q;
  }
  public void run() {
    Message msg ;
    try {
      // consuming messages until exit message is received
      while((msg = queue.take()).getMsg()!="exit") {
        Thread.sleep(10);
        System.out.println(Thread.currentThread().getName()+
                           " consumed "+msg.getMsg());
      }
      System.out.println("Consumer finished");
    } catch (InterruptedException e) { }
  }
}
```

Producer-Consumer: BlockingQueue

- Cosa succede se aggiungiamo un Consumer ?
- Ovviamente solo un consumer leggerà il messaggio «exit».
- Bisogna che il produttore emetta tanti «exit» quanti sono i consumatori
  - O che ci siano tanti produttori quanti consumatori.