



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



Il problema della coordinazione di più Thread

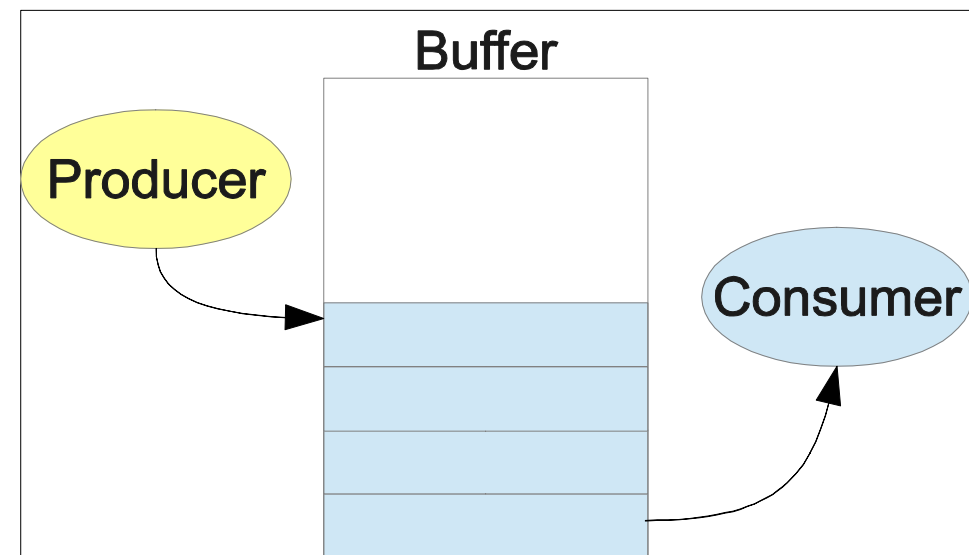
- Un problema che si verifica frequentemente in programmazione concorrente è l'esigenza di coordinare le attività dei Thread.
- Ad esempio:
 - ▶ Un programma può richiedere a due o più thread un'esecuzione di un determinato compito a turno.
 - ▶ Quando eseguire ciascun thread dipende dalla logica applicativa, quindi l'alternanza tra thread va controllata



PRODUTTORE CONSUMATORE

Problema del produttore-consumatore

- Un thread (il produttore) deposita dati in una zona di memoria condivisa (buffer)
- Un altro thread (il consumatore) preleva i dati dal buffer
- Il buffer è (ovviamente) limitato
- I due task producono e consumano continuamente, ma con frequenza variabile e non nota a priori.



- Il problema è assicurare che
 - ▶ il produttore non cerchi di inserire nuovi dati quando il buffer è pieno
 - ▶ il consumatore non cerchi di estrarre dati quando il buffer è vuoto.

Produttore-Consumatore: la soluzione

- Il produttore deve sospendere la propria esecuzione se il buffer è pieno
 - ▶ In attesa che non sia più pieno
- Il consumatore si sospende se il buffer è vuoto
 - ▶ In attesa che non sia più 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 Cella {  
    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;  
    }  
}
```

Iniziamo con un buffer di capienza unitaria.
La cella è thread-safe (è un monitor)



Esempio di implementazione errata

- Abbiamo a disposizione una classe thread-safe che implementa la Cella dati da condividere
 - ▶ In modo che i task produttore e consumatore non debbano preoccuparsi delle corse critiche
 - ▶ Cioè, ciascuno dei due viene messo in attesa quando l'altro sta modificando il buffer (o cella)
- Vediamo il codice dei task e il main.



Esempio di implementazione errata

```
import java.util.concurrent.ThreadLocalRandom;
class Produttore extends Thread{
    Cella cellaCondivisa;
    public Produttore(Cella c){
        this.cellaCondivisa=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)(ThreadLocalRandom.current().
                nextInt(0, 100));
            cellaCondivisa.setValore(v);
        }
    }
}
```



Esempio di implementazione errata

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

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



Esempio di implementazione errata

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

Esempio di implementazione errata

Un possibile output

- Chiaramente errato:
 - ▶ Inizia il consumatore (che consuma da buffer vuoto)
 - ▶ Diversi elementi sono sovrascritti (quindi persi).
 - ▶ Diversi elementi sono letti più volte

```
Console x
<terminated> ProdCons (14) [Java A
Viene letto 0
Viene scritto 7
Viene scritto 70
Viene letto 70
Viene scritto 50
Viene scritto 76
Viene letto 76
Viene letto 76
Viene scritto 83
Viene letto 83
Viene scritto 39
Viene letto 39
Viene scritto 97
Viene letto 97
Viene scritto 84
Viene letto 84
Viene scritto 55
Viene scritto 24
Viene letto 24
Viene letto 24
```



Esempio di implementazione corretta

- Il problema dell'implementazione vista è che non ci si preoccupava minimamente dello stato della cella.
- Scriviamo la classe Cella in modo che produzione e consumo procedano come desiderato.
 - ▶ Oltre a garantire accesso esclusivo alla cella.
- La cella blocca le scritture se piena
- La cella blocca le letture se vuota
- Ogni scrittura sblocca le letture
- Ogni lettura sblocca le scritture
- NB: tutta la responsabilità della sincronizzazione è nella cella condivisa.



Esempio di implementazione corretta

```
public class Cella {  
    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() {  
    while (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) {  
    while (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

Un possibile output

- L'output è corretto:
 - ▶ Inizia il produttore
 - ▶ Produzione/consumo alternati
 - Nessun elemento sovrascritto
 - Nessun elemento letto più volte

```
Console
<terminated> ProdCons (5) [Java
Scritto 45
Letto 45
Scritto 34
Letto 34
Scritto 58
Letto 58
Scritto 52
Letto 52
Scritto 98
Letto 98
Scritto 13
Letto 13
Scritto 77
Letto 77
Scritto 58
Letto 58
Scritto 23
Letto 23
Scritto 31
Letto 31
```



Codice “parlante”

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

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


Appena entrati, e comunque
prima di aver fatto modifiche

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 ");  
    while(numItems==0){  
        outState(" going to wait on get ");  
        wait();  
        outState(" going to notify in get ");  
        notify();  
    }  
    numItems--;  
    outState(" exiting get");  
    return valore;  
}
```





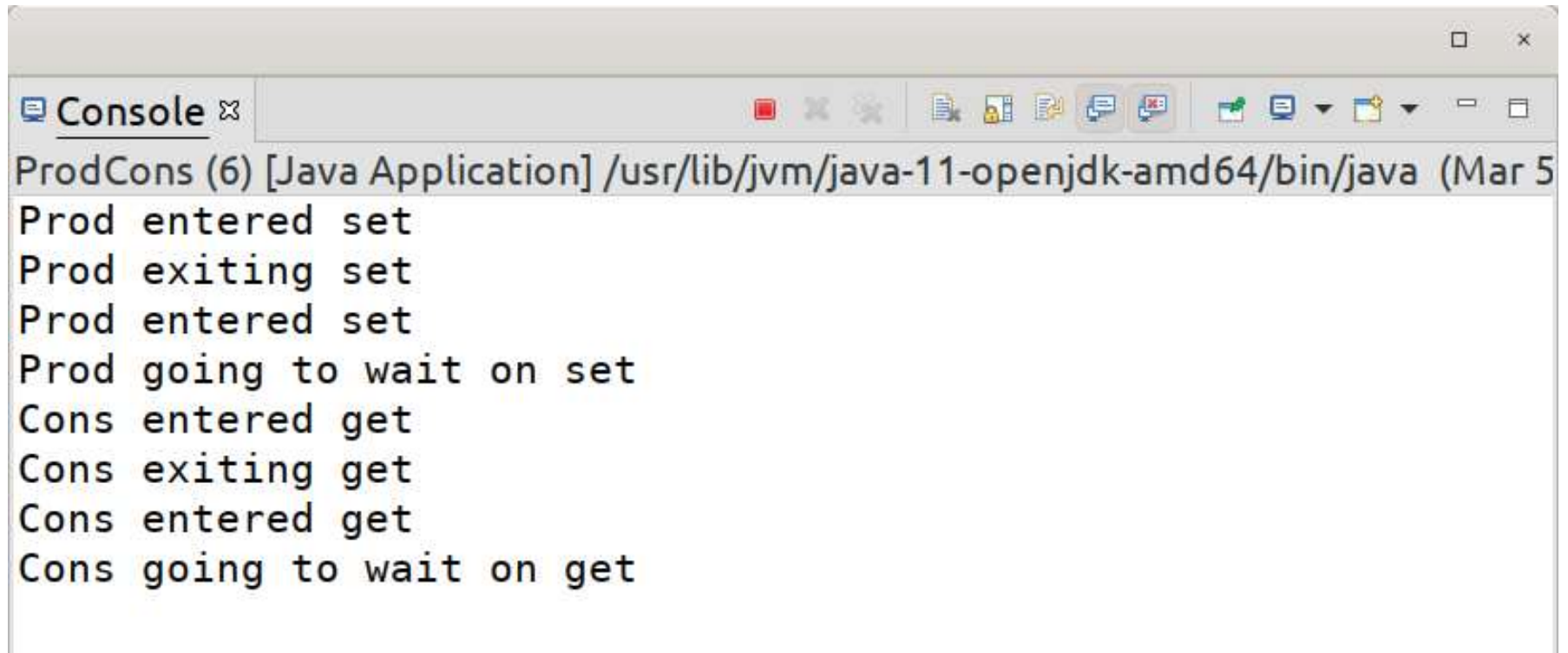
Esempio di implementazione errata

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



Esempio di implementazione errata

- Esempio di output scorretto, con buffer di dimensioni non unitarie

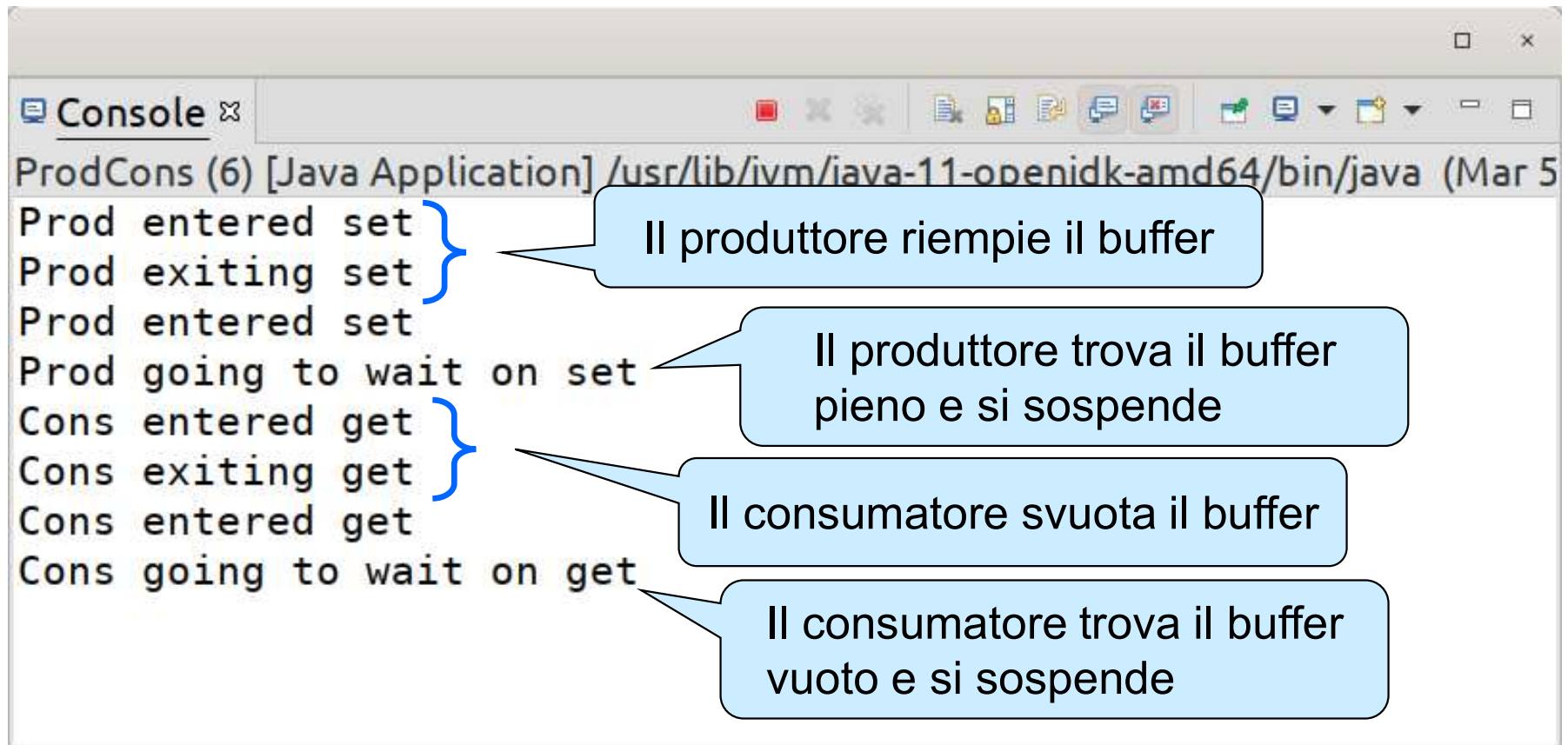


```
ProdCons (6) [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Mar 5
Prod entered set
Prod exiting set
Prod entered set
Prod going to wait on set
Cons entered get
Cons exiting get
Cons entered get
Cons going to wait on get
```

- Il sistema va in deadlock: perché?
- Per capirlo, sfruttiamo i metodi «parlanti» della cella

Esempio di implementazione errata

- Esempio di output scorretto [deadlock]



```
ProdCons (6) [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Mar 5
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
```

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

- Il produttore è in attesa
- Il consumatore anche

Osservazione

- Nella Cella, abbiamo trattato le condizioni nel modo discusso (con while):

```
while (numItems==0) {  
    try {  
        wait();  
    } catch (InterruptedException e) { }  
}  
while (numItems==BUFFERSIZE) {  
    try {  
        wait();  
    } catch (InterruptedException e) { }  
}
```

- Se avessimo scritto `if(condizione) { wait(); }` il programma funzionerebbe ugualmente bene, perché il buffer di dimensioni unitarie non permette due produzioni o due consumi consecutivi.



Altra osservazione

- Abbiamo usato `notify`
- Cosa cambierebbe usando `notifyAll`?
- Nulla: il produttore ha comunque un unico consumatore da svegliare, e viceversa.



Usiamo un buffer più grande

- Finora il buffer `Cella` conteneva un solo elemento.
- Vediamo che succede se ingrandiamo il buffer

Usiamo una coda

- In generale
 - ▶ Il buffer ha dimensioni maggiori di uno
 - ▶ Possono essere molti produttori e molti consumatori
 - ▶ Gli elementi vengono consumati nello stesso ordine in cui sono prodotti.
- Abbiamo bisogno di una coda (cioè di un buffer FIFO)
- Cominciamo a vedere una coda adatta all'uso in programmi non concorrenti



Coda

```
public class Coda {
    static int BUFFERSIZE;
    private int numItems = 0;
    private int[] valori;
    private int first; // index of the first item to read
    private int last; // index of the
                      // most recently inserted item

    Coda(int size) {
        BUFFERSIZE=size;
        first=0; last=0;
        valori=new int[BUFFERSIZE];
    }

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



Coda

```
public int getItem() {  
    int tmp;  
    if (numItems==0) {  
        System.err.print("lettura di buffer vuoto!\n");  
        System.exit(0);  
    }  
    numItems--;  
    tmp=valori[first];  
    first=(first+1)%BUFFERSIZE;  
    System.out.println("letto ", tmp);  
    return tmp;  
}
```



Coda

```
public void setItem(int v) {  
    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);  
}  
  
}
```



Coda thread safe e bloccante

- Dobbiamo rendere la coda adatta all'uso concorrente
 - ▶ Thread-safe
 - ▶ Deve bloccare i produttori quando la coda è piena
 - ▶ Deve bloccare i consumatori quando la coda è vuota



Coda thread safe e bloccante

```
public class Coda {
    final int BUFFERSIZE;
    private int numItems = 0;
    private int[] valori;
    private int first; // index of the first item to read
    private int last; //index of most recently inserted item
    Coda(int size) {
        BUFFERSIZE=size;
        first=0; last=0;
        valori=new int[BUFFERSIZE];
    }
    void printWithName(String s, int v) {
        String threadName=Thread.currentThread().getName();
        System.out.println(threadName+s+v+"["+numItems+"]");
    }
    synchronized public int getCurrentSize() {
        return numItems;
    }
}
```

`printWithName` viene chiamato solo da metodi `synchronized`, quindi è comunque eseguito in mutua esclusione.



Coda thread safe e bloccante

```
synchronized public int getItem() {  
    int tmp;  
    while (numItems==0) {  
        try {  
            wait();  
        } catch (InterruptedException e) { }  
    }  
    numItems--;  
    tmp=valori[first];  
    first=(first+1)%BUFFERSIZE;  
    printWithName(" letto ", tmp);  
    notifyAll();  
    return tmp;  
}
```



Coda thread safe e bloccante

```
synchronized public void setItem(int v) {  
    while (numItems==BUFFERSIZE) {  
        try {  
            wait();  
        } catch (InterruptedException e) { }  
    }  
    valori[last]=v;  
    last=(last+1)%BUFFERSIZE;  
    numItems++;  
    printWithName(" scritto ", v);  
    notifyAll();  
}  
  
}
```



Produttore

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

public class Produttore extends Thread {
    Coda buffer;
    public Produttore(String s, Coda c){
        super(s);
        this.buffer=c;
    }
    public void run(){
        int i=0;
        for(;;){
            try {
                Thread.sleep(ThreadLocalRandom.current().nextInt(10,100));
            } catch (InterruptedException e) { }
            buffer.setItem(i++);
        }
    }
}
```



Consumatore

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

public class Consumatore extends Thread {
    Coda buffer;
    int v;
    public Consumatore(String s, Coda c) {
        super(s);
        this.buffer=c;
    }
    public void run() {
        for(;;) {
            v=buffer.getItem();
            try {
                Thread.sleep(ThreadLocalRandom.current().nextInt(10,100));
            } catch (InterruptedException e) { }
        }
    }
}
```



main

```
public class ProdCons {
    final int BUFFSIZE=4;
    Coda coda=new Coda(BUFFSIZE);
    void exec(){
        new Produttore("Prod1", coda).start();
        new Consumatore("Cons1", coda).start();}
    public static void main(String[] args) {
        new ProdCons().exec();
    }
}
```

Possibile output

- Possibile output:

Cons1 letto 4[0]
Prod1 scritto 5[1]
Cons1 letto 5[0]
Prod1 scritto 6[1]
Cons1 letto 6[0]
Prod1 scritto 7[1]
Prod1 scritto 8[2]
Cons1 letto 7[1]
Prod1 scritto 9[2]
Cons1 letto 8[1]
Prod1 scritto 10[2]
Prod1 scritto 11[3]
Cons1 letto 9[2]
Prod1 scritto 12[3]
Prod1 scritto 13[4]
Cons1 letto 10[3]
Prod1 scritto 14[4]
Cons1 letto 11[3]

Ci sono più scritture e letture consecutive:
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



main

```
public class ProdCons {  
    final int BUFFSIZE=4;  
    Coda coda=new Coda(BUFFSIZE);  
    void exec() {  
        Coda cella=new Coda(4);  
        new Produttore("Prod1", coda).start();  
        new Consumatore("Cons1", coda).start();  
        new Produttore("Prod2", coda).start();  
        new Consumatore("Cons2", coda).start();  
    }  
    public static void main(String[] args) {  
        new ProdCons().exec();  
    }  
}
```

Tanti thread che producono
Tanti thread che consumano
Tutti usando il medesimo buffer,
che ha capienza > 1

Possibile output

- Possibile output:

Prod2 scritto 7[1]

Prod1 scritto 6[2]

Prod2 scritto 8[3]

Prod1 scritto 7[4]

Cons2 letto 7[3]

Cons1 letto 6[2]

Prod2 scritto 9[3]

Cons1 letto 8[2]

Cons2 letto 7[1]

Prod1 scritto 8[2]

Prod2 scritto 10[3]

Prod1 scritto 9[4]

Cons2 letto 9[3]

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

Osservazioni

- Coda fa `notifyAll()` dopo letture e scritture
- In questo modo si svegliano **tutti** i thread bloccati sull'oggetto.
 - ▶ Un produttore che abbia appena riempito il buffer sveglia altri produttori che erano in attesa che il buffer fosse non pieno
 - ▶ Un consumatore che abbia appena svuotato il buffer sveglia altri consumatori che erano in attesa che il buffer fosse non vuoto
 - ▶ Un consumatore che abbia appena reso il buffer non pieno sveglia molti produttori, anche se poi ne basta uno a riempire nuovamente il buffer
 - ▶ ...
- Questo non provoca problemi?



Osservazioni

- Non c'è alcun problema
 - ▶ perché tutti i thread tornano poi a valutare la condizione per andare avanti: molti si ri-bloccheranno subito
- NB: per questo è essenziale che la valutazione della condizione sia effettuata con **while (condizione) { wait(); }**



Qual è l'effetto di `notifyAll`?

- Quando si fa `notifyAll`, vengono risvegliati tutti i thread in attesa sul buffer, 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.



Coda

```
import java.util.concurrent.Semaphore;

public class Coda {
    static int BUFFERSIZE;
    private int numItems = 0;
    private int[] valori;
    private int first; // index of the first item to read
    private int last; // index of the most recently inserted item
    Semaphore mutex;
    Coda(int size){
        BUFFERSIZE=size;
        mutex = new Semaphore(1);
        first=0; last=0;
        valori=new int[BUFFERSIZE];
    }
}
```




Coda

```
void printWithName(String s, int v) {  
    String threadName=Thread.currentThread().getName();  
    System.out.println(threadName+s+v+"["+numItems+"]");  
}  
public int getCurrentSize(){  
    return numItems;  
}
```



Coda

```
public int getItem() {
    int tmp;
    try{
        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);
    mutex.release();
    return tmp;
}
```



Coda

```
public void setItem(int v) {  
    try{  
        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);  
    mutex.release();  
}  
}
```



main

```
import java.util.concurrent.Semaphore;
public class ProdCons {
    final int BUFSIZE=4;
    final Semaphore canConsume = new Semaphore(0);
    final Semaphore canProduce = new Semaphore(BUFSIZE);
    void exec() {
        Coda cella=new Coda(BUFSIZE);
        new Produttore("Prod1", cella,
                       canConsume, canProduce).start();
        new Consumatore("Cons1", cella,
                       canConsume, canProduce).start();
        new Produttore("Prod2", cella,
                       canConsume, canProduce).start();
        new Consumatore("Cons2", cella,
                       canConsume, canProduce).start();
    }
    public static void main(String[] args) {
        new ProdCons().exec();
    }
}
```



Consumatore

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

public class Consumatore extends Thread {
    Coda buffer;
    int v;
    Semaphore canConsume;
    Semaphore canProduce;
    public Consumatore(String s, Coda c,
        Semaphore canConsume, Semaphore canProduce) {
        super(s);
        this.buffer=c;
        this.canProduce=canProduce;
        this.canConsume=canConsume;
    }
}
```



Consumatore

```
public void run() {  
    for(;;) {  
        try{  
            canConsume.acquire();  
        } catch (InterruptedException e) {}  
        v=buffer.getItem();  
        canProduce.release();  
        try {  
            Thread.sleep(ThreadLocalRandom.current().  
                nextInt(10, 100));  
        } catch (InterruptedException e) {}  
    }  
}  
}
```



Produttore

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

public class Produttore extends Thread {
    Coda buffer;
    Semaphore canConsume;
    Semaphore canProduce;
    public Produttore(String s, Coda c,
        Semaphore canConsume, Semaphore canProduce) {
        super(s);
        this.buffer=c;
        this.canProduce=canProduce;
        this.canConsume=canConsume;
    }
}
```



Produttore

```
public void run() {
    int i=0;
    for(;;) {
        try {
            Thread.sleep(ThreadLocalRandom.current().
                nextInt(10, 100));
        } catch (InterruptedException e) { }
        try{
            canProduce.acquire();
        } catch (InterruptedException e) {}
        buffer.setItem(i++);
        canConsume.release();
    }
}
}
```




Esecuzione

- OK!

```
Console
<terminated> ProdCons (10) [Java Application]
Prod scritto 0[1]
Cons letto 0[0]
Prod scritto 1[1]
Cons letto 1[0]
Prod scritto 2[1]
Cons letto 2[0]
Prod scritto 3[1]
Cons letto 3[0]
Prod scritto 4[1]
Prod scritto 5[2]
Cons letto 5[1]
Prod scritto 6[2]
Prod scritto 7[3]
Cons letto 7[2]
Prod scritto 8[3]
Cons letto 8[2]
Cons letto 8[1]
Cons letto 8[0]
Prod scritto 9[1]
Prod scritto 10[2]
Cons letto 10[1]
Cons letto 10[0]
Prod scritto 11[1]
Cons letto 11[0]
Prod scritto 12[1]
Prod scritto 13[2]
```

Esempio di output (frammento)

Prod1 scritto 0[1]

Cons1 letto 0[0]

Prod2 scritto 0[1]

Cons2 letto 0[0]

Prod1 scritto 1[1]

Cons1 letto 1[0]

Prod2 scritto 1[1]

Cons2 letto 1[0]

Prod2 scritto 2[1]

Prod1 scritto 2[2]

Cons2 letto 2[1]

Prod1 scritto 3[2]

Cons2 letto 2[1]

Cons1 letto 3[0]

Prod1 scritto 4[1]

Cons1 letto 4[0]

[omissis]

Cons2 letto 6[0]

Prod2 scritto 7[1]

Cons1 letto 7[0]

Prod1 scritto 11[1]

Cons1 letto 11[0]

Prod2 scritto 8[1]

Cons2 letto 8[0]

Prod1 scritto 12[1]

Cons1 letto 12[0]

Prod2 scritto 9[1]

Prod1 scritto 13[2]

Prod2 scritto 10[3]

Cons2 letto 9[2]

Prod2 scritto 11[3]

Prod1 scritto 14[4]

Cons1 letto 13[3]

Prod2 scritto 12[4]

I dati sono consumati
nello stesso ordine con
cui sono stati prodotti
La coda contiene
sempre tra 0 e 4
elementi

Usiamo le librerie di Java

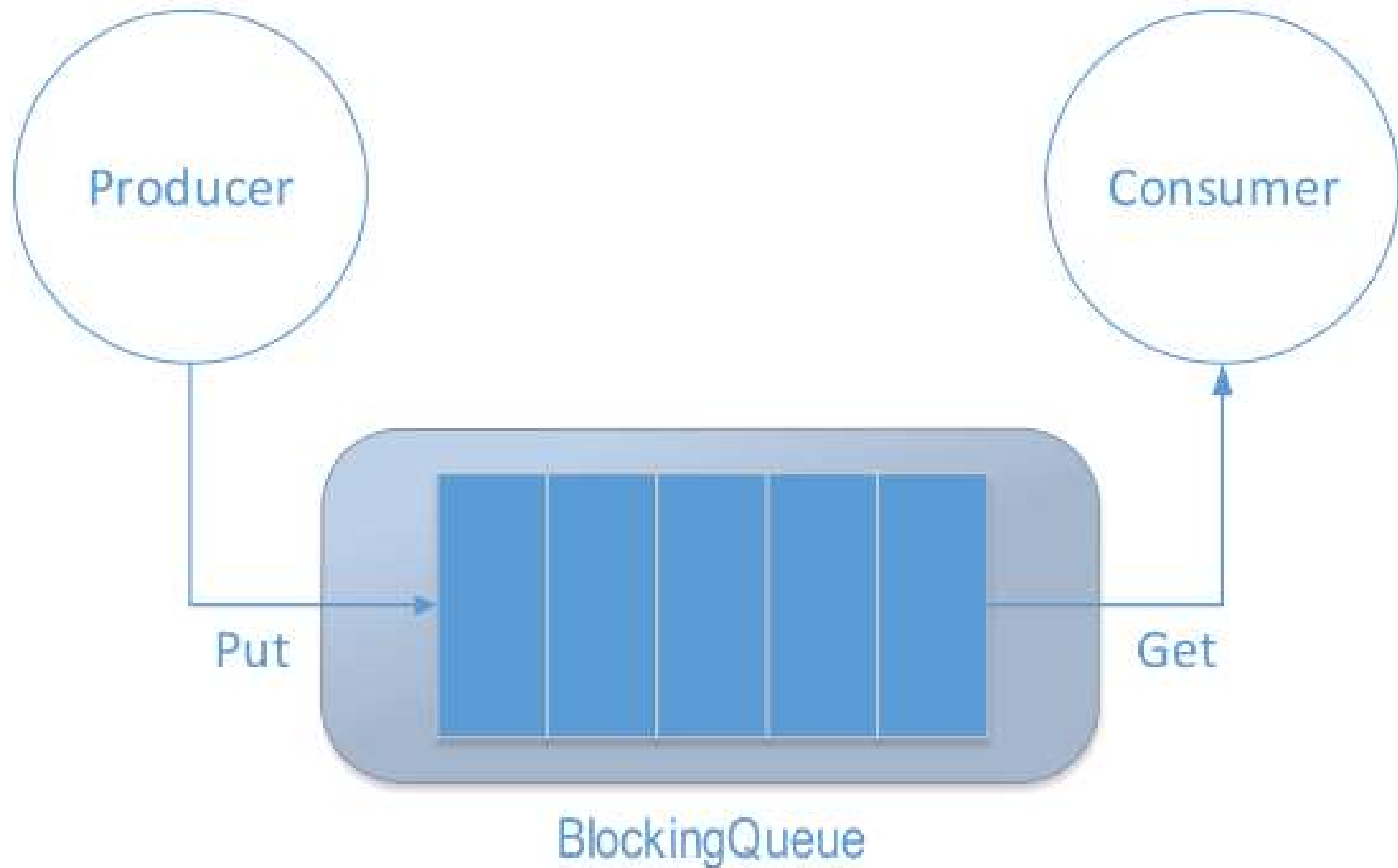
- Forniscono una coda thread-safe bloccante
 - ▶ La classe BlockingQueue
- Non dobbiamo preoccuparci noi di implementarla!
- L'interfaccia BlockingQueue è parte integrante del collections framework di java ed è principalmente utilizzata per la realizzazione di programmi del tipo produttore-consumatore.
- 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.



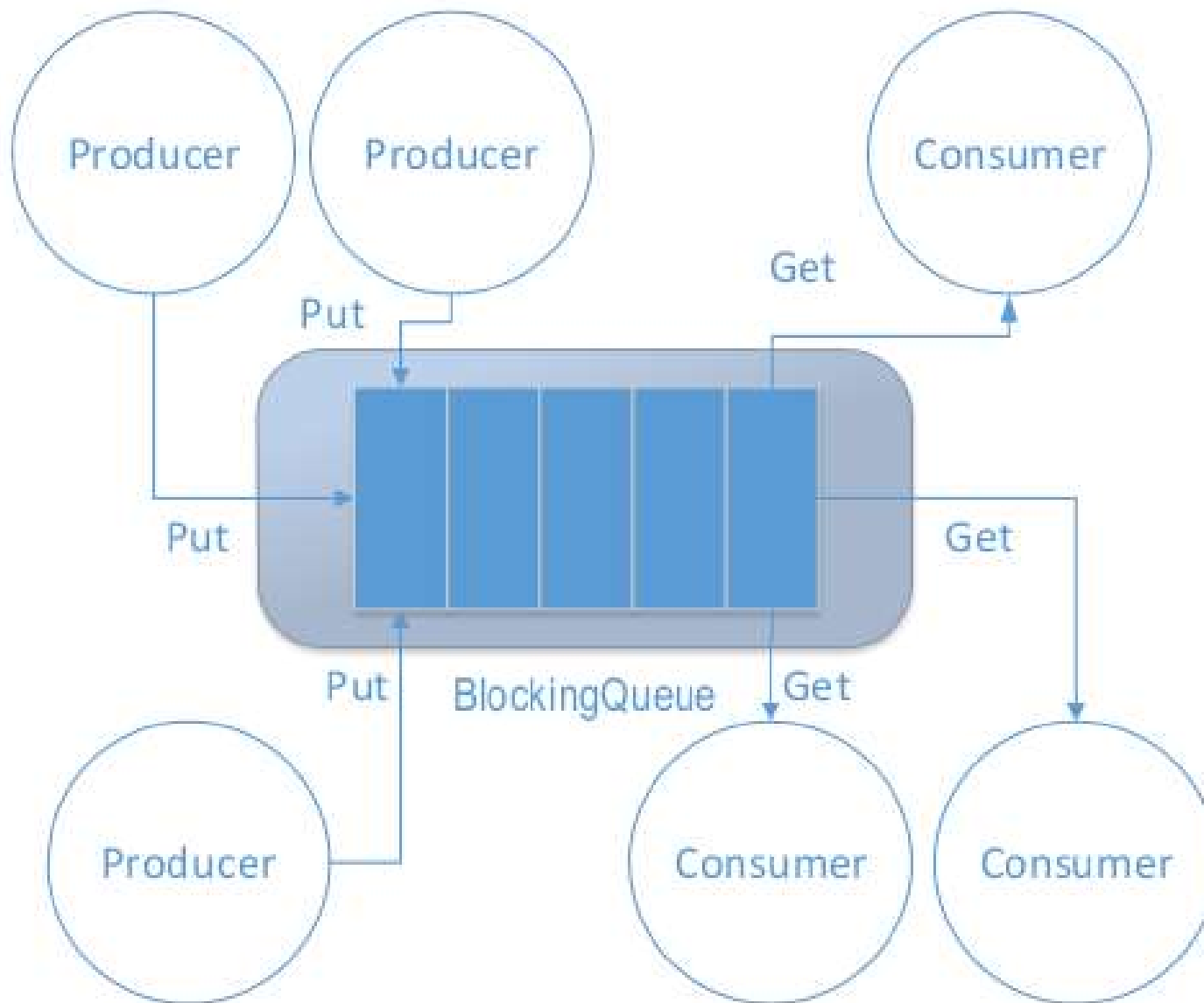
`java.util.concurrent: BlockingQueue`

- `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 expired



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.
- Come vedremo, **BlockingQueue** offre una varietà di metodi, anche non bloccanti

BlockingQueue: metodi

- Ci sono diversi metodi, sia bloccanti sia non bloccanti
- In caso di problemi, i metodi non bloccati
 - ▶ restituiscono un valore che indica il fallimento dell'operazione
 - ▶ sollevano un'eccezione

	<i>Throws exception</i>	<i>Special value</i>	<i>Blocks</i>	<i>Times out</i>
Insert	<code>add(e)</code>	<code>offer(e)</code>	<code>put(e)</code>	<code>offer(e, time, unit)</code>
Remove	<code>remove()</code>	<code>poll()</code>	<code>take()</code>	<code>poll(time, unit)</code>
Examine	<code>element()</code>	<code>peek()</code>	<i>not applicable</i>	<i>not applicable</i>



Producer-Consumer: BlockingQueue

- Useremo un **ArrayBlockingQueue** e i seguenti metodi
 - ▶ **void 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.

Metodi di una BlockingQueue

- Blocking methods:

Return type	method	description
void	put(E e)	Inserts the specified element into this queue, waiting if necessary for space to become available.
E	take()	Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

Metodi di una BlockingQueue

- Metodi che restituiscono valori specifici:

Return type	method	description
boolean	offer(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and false if no space is currently available
E	poll()	Retrieves and removes the head of this queue, or returns null if this queue is empty
E	peek()	Retrieves, but does not remove , the head of this queue, or returns null if this queue is empty



Metodi di una BlockingQueue

- Metodi bloccanti con timeout:

return	method	description
boolean	offer(E e, long timeout, TimeUnit unit)	Inserts the specified element into this queue, <u>waiting</u> up to the specified wait time if necessary for space to become available. Returns true if successful, or false if the specified waiting time elapses before space is available
E	poll(long timeout, TimeUnit unit)	Retrieves and removes the head of this queue, <u>waiting</u> up to the specified wait time if necessary for an element to become available. Returns null if the specified waiting time elapses before an element is available

Metodi di una BlockingQueue

- Metodi che sollevano eccezioni:

Return type	method	description
boolean	add(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available
E	remove()	Retrieves and removes the head of this queue. Throws NoSuchElementException if this queue is empty
E	element()	Retrieves, but does not remove , the head of this queue. This method throws a NoSuchElementException exception if this queue is empty.



Producer-Consumer con BlockingQueue

- Implementeremo una coda di messaggi (classe Message)



Message

```
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();
    }
}
```



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() {
        String mioNome=Thread.currentThread().getName();
        int i=0;
        while(true) {
            Message msg = new Message (mioNome+"_dato_" + i);
            try {
                Thread.sleep(10);
                queue.put(msg);    // produce messages
                System.out.println(mioNome+" produced "+msg.getMsg()+
                                   "["+queue.size()+"]");
            } catch (InterruptedException e) { }
        }
    }
}
```



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 {
            while(true) {
                msg = queue.take() ;
                System.out.println(Thread.currentThread().getName() +
                    " consumed " + msg.getMsg() + "[" + queue.size() + "]" );
                Thread.sleep(10) ;
            }
        } catch (InterruptedException e) { }
    }
}
```