



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

Programmazione Concorrente e Distribuita Paradigmi di comunicazione

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



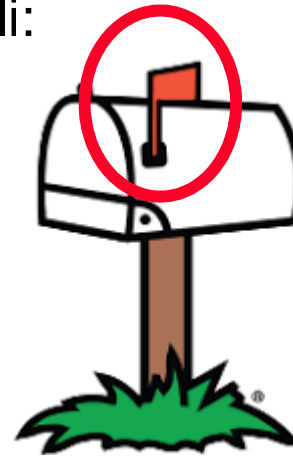
Paradigmi di comunicazione

- Ci sono tanti modi con cui i thread possono comunicare tra loro.
- Ne vedremo alcuni tra i più comuni:
 - ▶ Signals
 - ▶ Buffer
 - ▶ Blackboard
 - ▶ Broadcast
 - ▶ Barrier

Signal

- Consente ad un thread di attendere un segnale inviato da un altro thread
- Ci sono tradizionalmente due tipi di segnali:

- ▶ Persistent signal: è un segnale che rimane impostato fino a quando un singolo thread non lo riceve.



- ▶ Transient signal: è un segnale che rilascia uno o più thread in attesa, ma si perde se non ci sono thread in attesa.





Signal

- Due ruoli ben distinti:
 - ▶ Chi aspetta un segnale
 - ▶ Chi invia il segnale atteso.
- Spesso si tratta di thread distinti
- ➔ separiamo l'interfaccia del Thread che invia da quello che aspetta
 - ▶ In questo modo ogni Thread può chiamare l'operazione più appropriata



Signal: interfacce

```
public interface SignalSender {  
    void send();  
}
```

```
public interface SignalWaiter {  
    void waits() throws InterruptedException;  
}
```

Se si vuole evitare un'attesa infinita, si può aggiungere un metodo **waits(t)** con un time-out



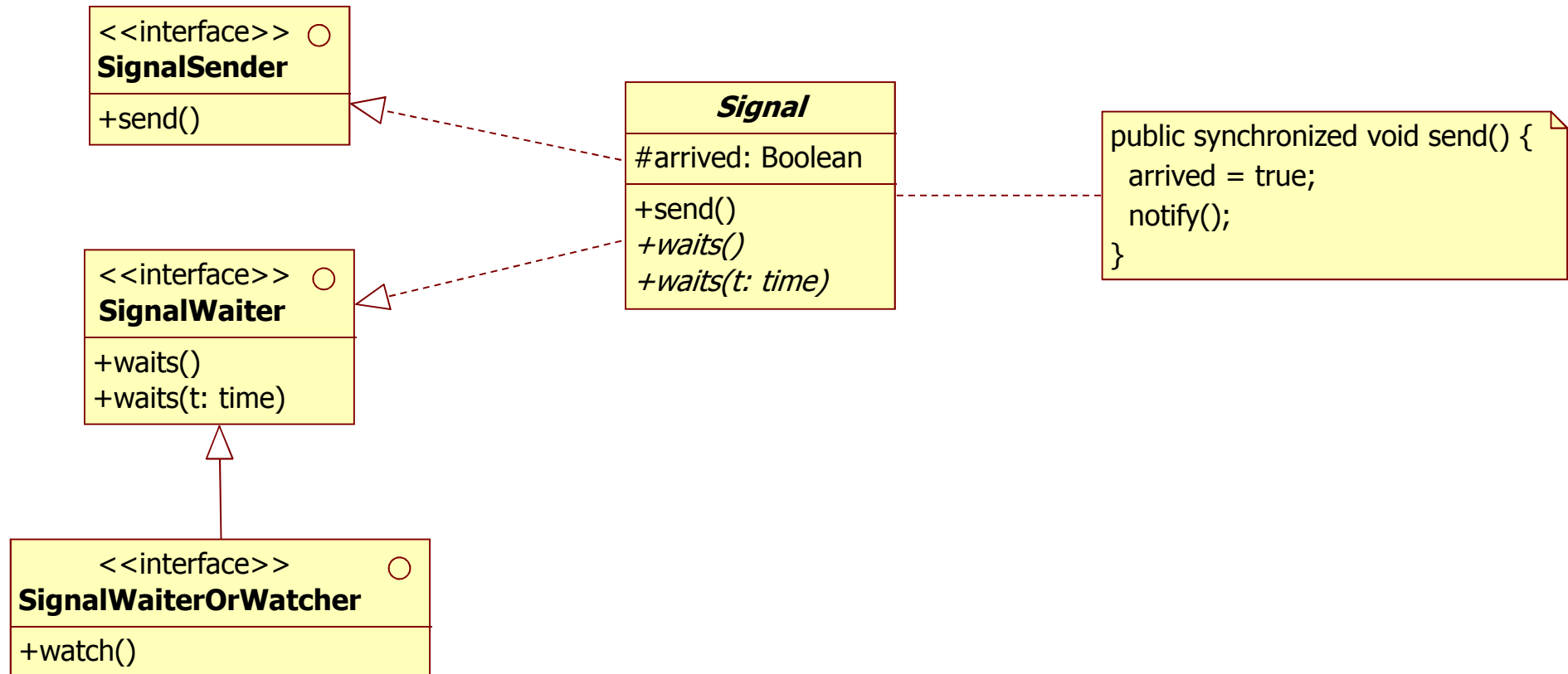
Class Signal

Definiamo una classe astratta dalla quale potremo derivare classi concrete che realizzano i segnali persistenti e transienti.

```
public abstract class Signal implements SignalSender,  
                                         SignalWaiter {  
    protected boolean arrived = false;  
    public abstract void waits() throws InterruptedException;  
    public synchronized void send() {  
        arrived = true;  
        notify();  
    }  
}
```



Signal: interfacce e classe astratta





Persistent Signal

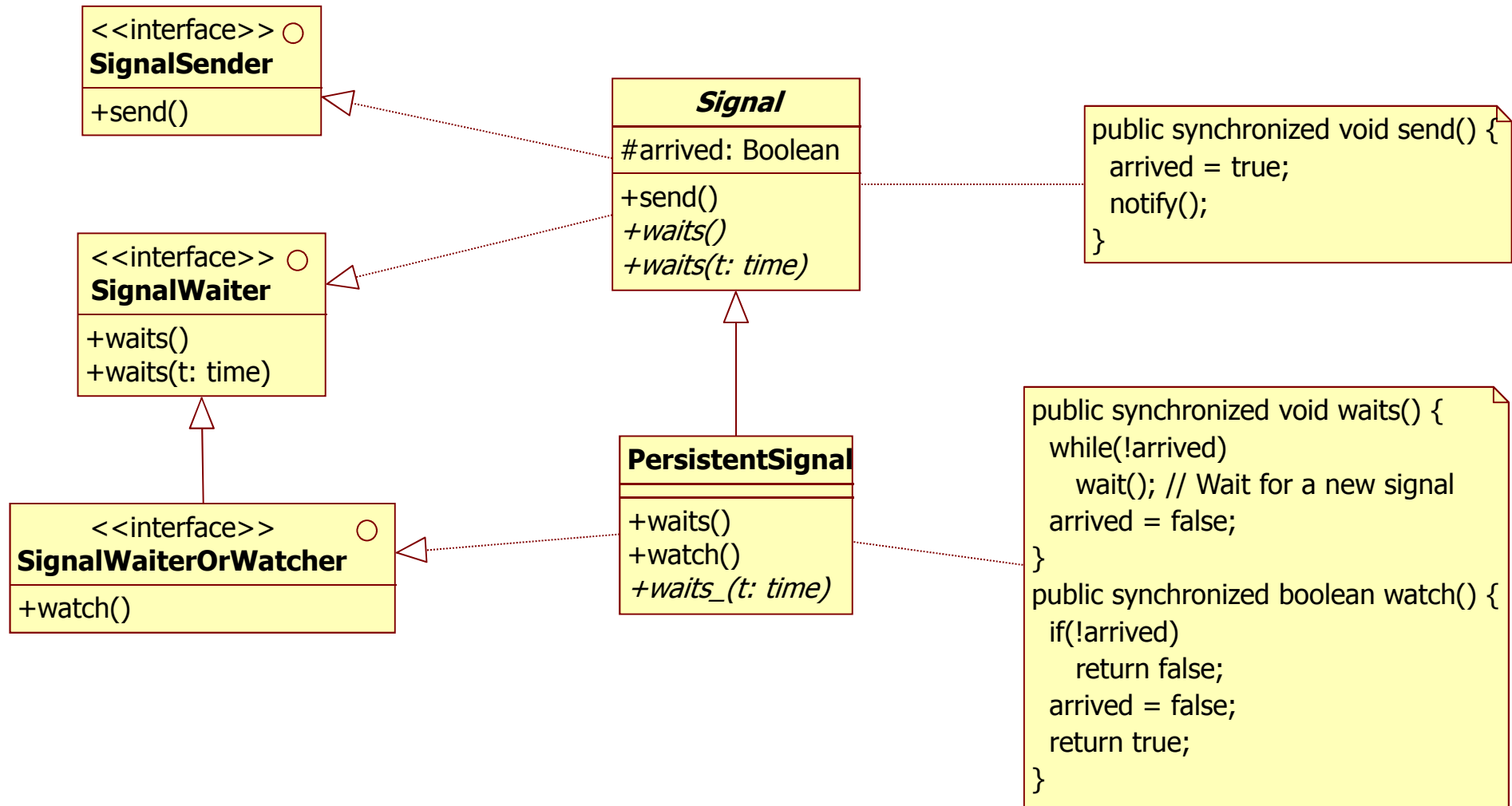
```
public interface SignalWaiterOrWatcher extends SignalWaiter {
    boolean watch();
}

public class PersistentSignal extends Signal
    implements SignalWaiterOrWatcher {
    public synchronized void waits()
        throws InterruptedException {
        while(!arrived)
            wait(); // Wait for a new signal
        arrived = false;
    }
    public synchronized boolean watch() {
        // This method never waits
        if(!arrived)
            return false;
        arrived = false;
        return true;
    }
}
```

Per sospendersi in attesa dell'evento

Per fare polling

PersistentSignal





Esempio di Persistent Signal: accesso al disco

```
public class DiskController {  
    private PersistentSignal reply;  
  
    // Various operations including:  
  
    public SignalWaiterOrWatcher asyncWrite(int blockNumber,  
                                             Block from) {  
        reply=new PersistentSignal();  
        // Set up the write operation then:  
        return reply;  
    }  
}
```

Fa return prima di aver concluso la scrittura.
E' il chiamante che deve verificare (usando
reply) se la scrittura va a buon fine o no.



Esempio di Persistent Signal: accesso al disco

Soluzione 1

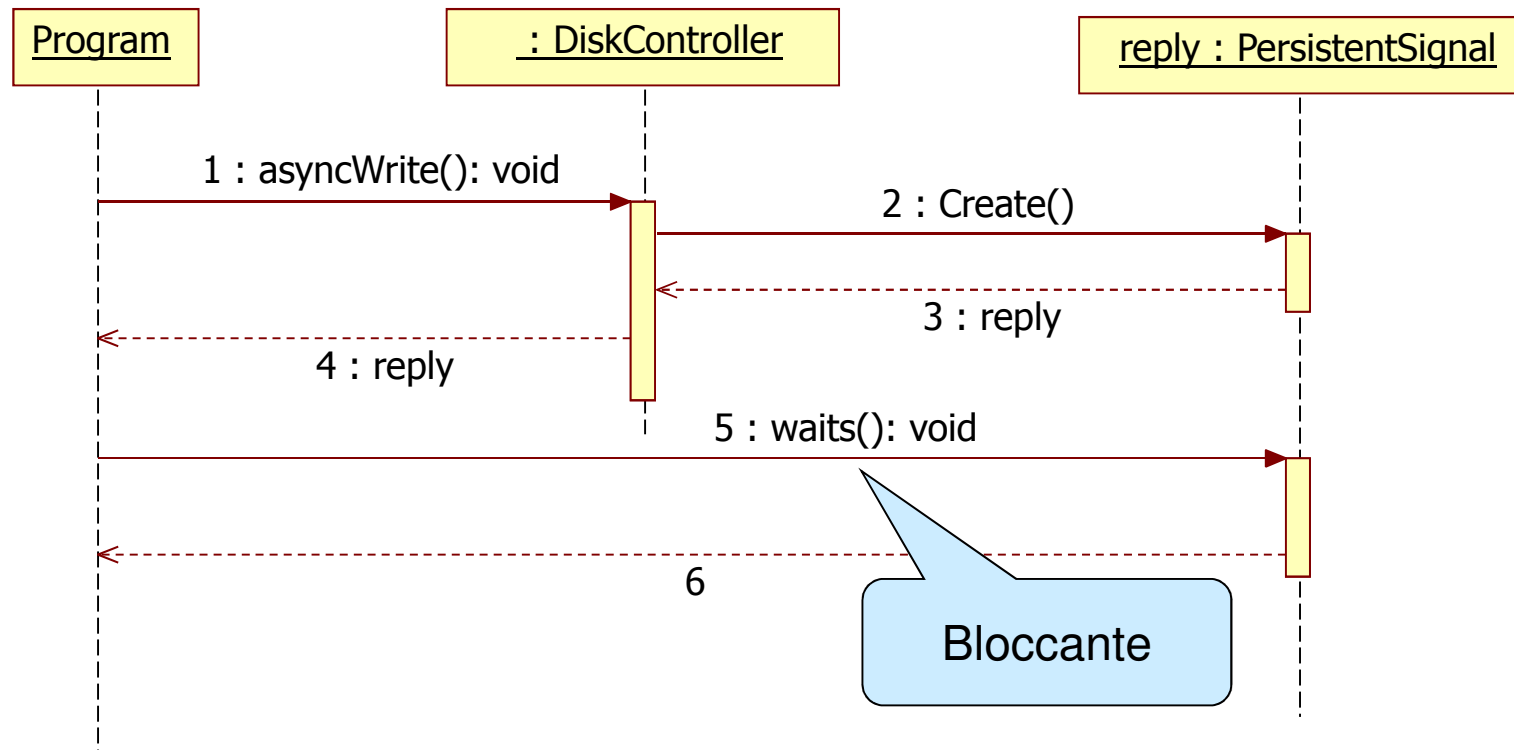
```
public static void main(String[] args) {  
    DiskController controller = new DiskController();  
    Block superBlock = new Block();  
    SignalWaiterOrWatcher outputDone;  
    outputDone = controller.asyncWrite(0, superBlock);  
    // . . . .  
    // When it is time to check that the output is complete:  
    try {  
        outputDone.waits();  
    } catch (InterruptedException ie )  
        // Initiate recovery action.  
    }  
}
```

Aspetta che l'operazione di scrittura sia terminata o che succeda qualcosa (InterruptedException).



Esempio di Persistent Signal: accesso al disco

Soluzione 1





Esempio di Persistent Signal: accesso al disco

Soluzione 2

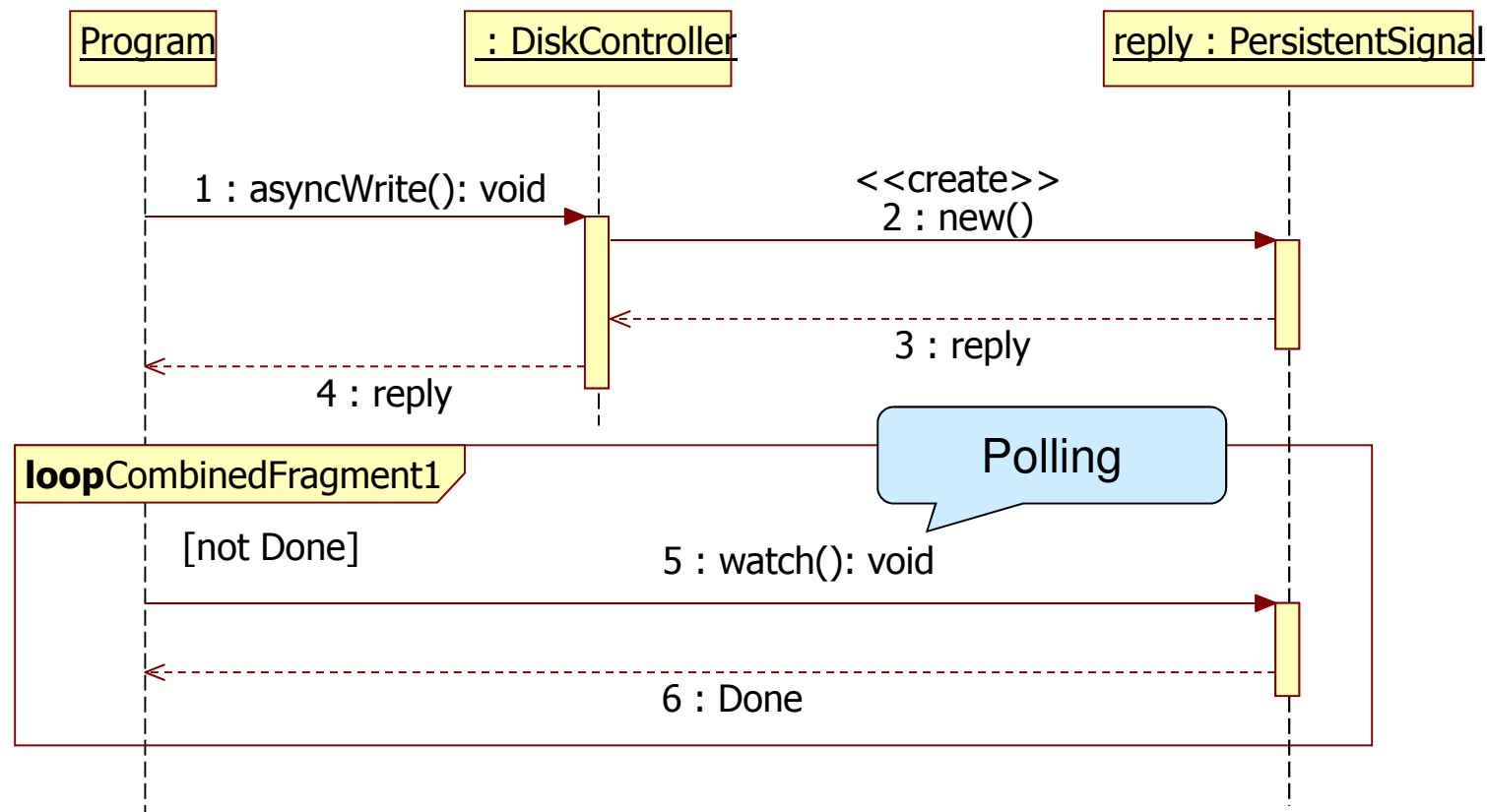
```
public static void main(String[] args) {  
    DiskController controller = new DiskController();  
    Block superBlock = new Block();  
    SignalWaiterOrWatcher outputDone;  
    outputDone = controller.asyncWrite(0, superBlock);  
    // . . . .  
    // When it is time to check that the output is complete:  
    if(!outputDone.watch()){  
        // Output not complete, initiate recovery action.  
    }  
}
```

Non aspetta ulteriormente che l'operazione di scrittura sia terminata: inizia subito recovery.



Esempio di Persistent Signal: accesso al disco

Soluzione 2





Gestiamo le attese

```
public interface SignalWaiter {  
    void waits() throws InterruptedException;  
    boolean waits(long t) throws InterruptedException;  
}
```

Aggiungiamo la **waits** con timeout.



Gestiamo le attese

```
public class PersistentSignal extends Signal
    implements SignalWaiterOrWatcher {
    public synchronized boolean waits(long t)
        throws InterruptedException {
        long call_time=System.currentTimeMillis(), elapsed=0;
        while(!arrived && elapsed < t){
            wait(t-elapsed); // Wait for signal
            if(arrived) {
                arrived = false;
                return true;
            } else {
                elapsed=System.currentTimeMillis()-call_time;
            }
        }
        return false;
    }
}
```




Esempio di Persistent Signal: accesso al disco

Soluzione 3

```
public static void main(String[] args) {
    DiskController controller = new DiskController();
    Block superBlock = new Block();
    SignalWaiterOrWatcher outputDone;
    System.out.println("Starting Write operation");
    outputDone = controller.asyncWrite(0, superBlock);
    // . . . .
    // When it is time to check that the output is complete:
    try {
        if(outputDone.waits((long) 345)) {
            System.out.println("Write completed");
        } else {
            System.out.println("Write not completed");
        }
    } catch (InterruptedException ie ) {
        // Initiate recovery action.
    }
}
```

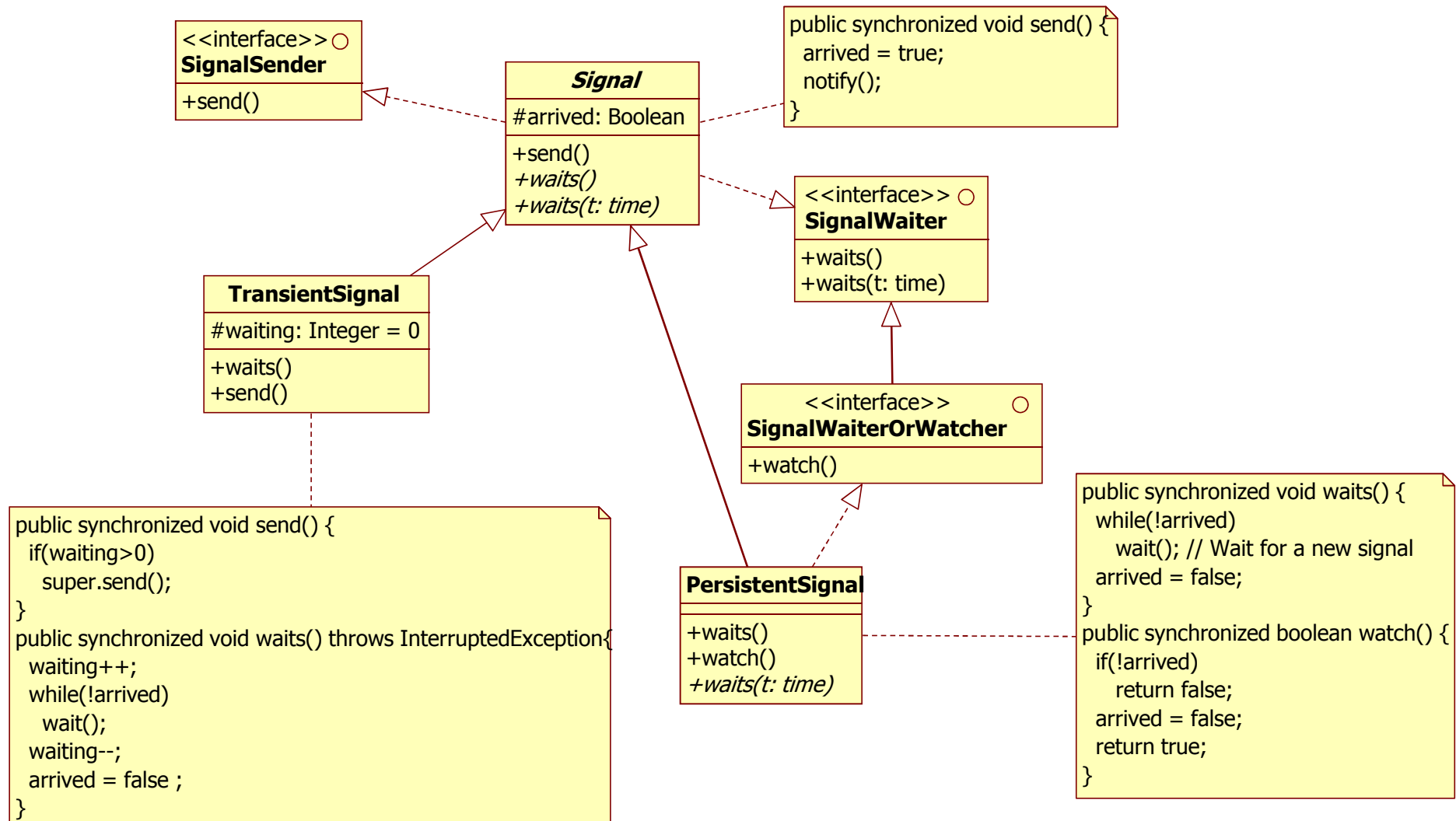


Transient Signal

```
public class TransientSignal extends Signal {
    protected int waiting = 0;
    public synchronized void send() {
        if(waiting>0)
            super.send();    // Sveglia un solo Thread
    }
    public synchronized void waits() throws InterruptedException{
        try {
            waiting++;
            while(!arrived)
                wait();
            waiting--;
            arrived = false ;
        } catch (InterruptedException ie) {
            waiting--;
            throw ie;
        }
    }
}
```

NB: no watch!

TransientSignal



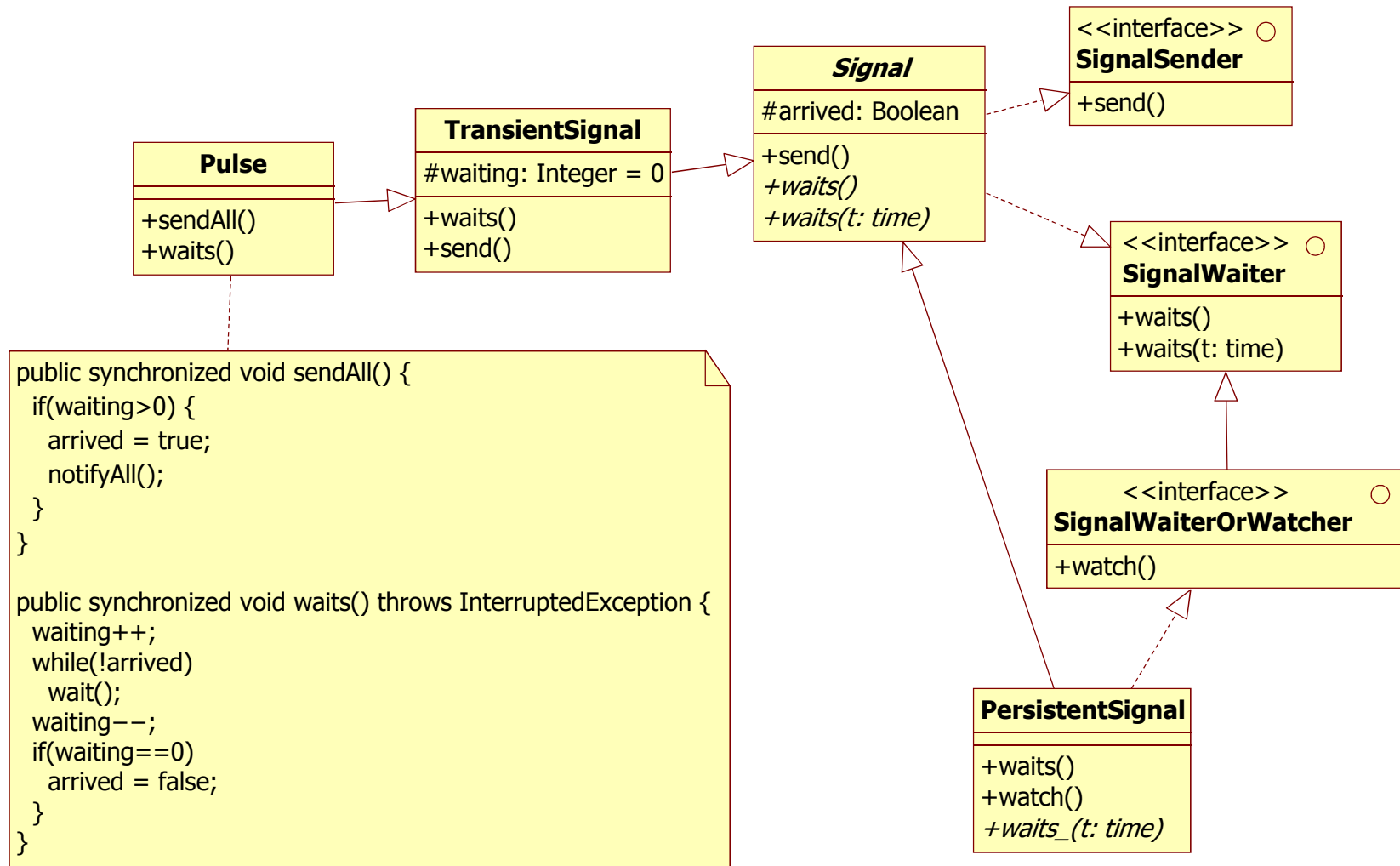


Class Pulse: rilascia tutti i Thread in waitForSignal

```
public class Pulse extends TransientSignal {  
    // Sveglia tutti i thread in attesa del segnale  
    public synchronized void sendAll() {  
        if(waiting>0) {  
            arrived = true;  
            notifyAll();  
        }  
    }  
}
```

Sveglia tutti i thread in attesa, mentre la classe base ne sveglia uno solo.

Pulse





Class Pulse:

rilascia tutti i Thread in waitForSignal

```
public synchronized void waits()  
    throws InterruptedException {  
    // Overrides inherited waits  
    try {  
        waiting++;  
        while(!arrived)  
            wait();  
        waiting--;  
        if(waiting==0)  
            arrived = false;  
    } catch (InterruptedException ie) {  
        if(--waiting==0)  
            arrived = false;  
        throw ie;  
    }  
}
```

Solo l'ultimo thread
svegliato resetta il flag.



Esempio di uso di Signals: Attesa per Saldi

- Durante la stagione dei saldi un negozio decide di aprire le porte solo ad intervalli di tempo di N secondi
- I clienti che arrivano quando le porte sono chiuse aspettano
- Quando le porte vengono aperte, tutti i clienti in attesa entrano
- Le porte si richiudono subito dopo che i clienti sono entrati



Attesa per Saldi: class Cliente

```
public class Cliente extends Thread {
    private Pulse portaNegozio;
    public Cliente(Pulse pn) {
        portaNegozio=pn;
        setName("Cliente "+getName());
    }
    public void run() {
        System.out.println(getName()+" : mi metto in coda");
        try {
            portaNegozio.waits();
            System.out.println(getName()+" : entrato nel negozio");
        } catch (InterruptedException e) {}
    }
}
```




Attesa per Saldi: class Negozio

```
public class Negozio extends Thread {
    private Pulse portaNegozio;
    public Negozio(Pulse pn) {
        portaNegozio=pn;
        setName("Negozio" + getName());
    }
    public void run() {
        while(true) {
            try {
                Thread.sleep(1000);
                System.out.println(getName() + ": apro le porte");
                portaNegozio.sendAll();
                System.out.println(getName() + ": chiudo le porte");
            } catch (InterruptedException e) { }
        }
    }
}
```



Attesa per Saldi: class Saldi

```
public class Saldi {
    public static void main(String[] args) {
        Pulse portaNegozio=new Pulse();
        new Negozio(portaNegozio).start();
        while(true) {
            int numClienti=(int)(5* Math.random());
            for(int i=0; i<numClienti; i++) {
                new Cliente(portaNegozio).start();
            }
            try {
                Thread.sleep(300);
            } catch (InterruptedException e) {}
        }
    }
}
```



Esempio di output

Cliente Thread-1: mi metto in coda
Cliente Thread-4: mi metto in coda
Cliente Thread-3: mi metto in coda
Cliente Thread-2: mi metto in coda
NegozioThread-0: apro le porte
Cliente Thread-2: sono entrato nel negozio
Cliente Thread-1: sono entrato nel negozio
Cliente Thread-4: sono entrato nel negozio
Cliente Thread-3: sono entrato nel negozio
Cliente Thread-5: mi metto in coda
Cliente Thread-8: mi metto in coda
Cliente Thread-7: mi metto in coda
Cliente Thread-6: mi metto in coda
Cliente Thread-9: mi metto in coda
Cliente Thread-10: mi metto in coda
Cliente Thread-11: mi metto in coda
Cliente Thread-12: mi metto in coda
NegozioThread-0: apro le porte

Cliente Thread-12: sono entrato nel negozio
Cliente Thread-9: sono entrato nel negozio
Cliente Thread-10: sono entrato nel negozio
Cliente Thread-5: sono entrato nel negozio
Cliente Thread-11: sono entrato nel negozio
Cliente Thread-8: sono entrato nel negozio
Cliente Thread-7: sono entrato nel negozio
Cliente Thread-6: sono entrato nel negozio
Cliente Thread-13: mi metto in coda
Cliente Thread-14: mi metto in coda
Cliente Thread-16: mi metto in coda
Cliente Thread-15: mi metto in coda



Buffers

- Il buffer contiene dati che una volta letti verranno distrutti.
- Se i dati devono essere conservati, il paradigma Blackboard –che vedremo più avanti – è più appropriato.



Buffers: BoundedBuffer generico

```
public class BoundedBuffer<Data> {  
    private Data buffer[];  
    private int first, last;  
    private int numberInBuffer = 0;  
    private int size;  
    public BoundedBuffer(int s) {  
        size=s;  
        buffer=(Data[]) new Object[size];  
        last=0;  
        first=0;  
    }  
}
```

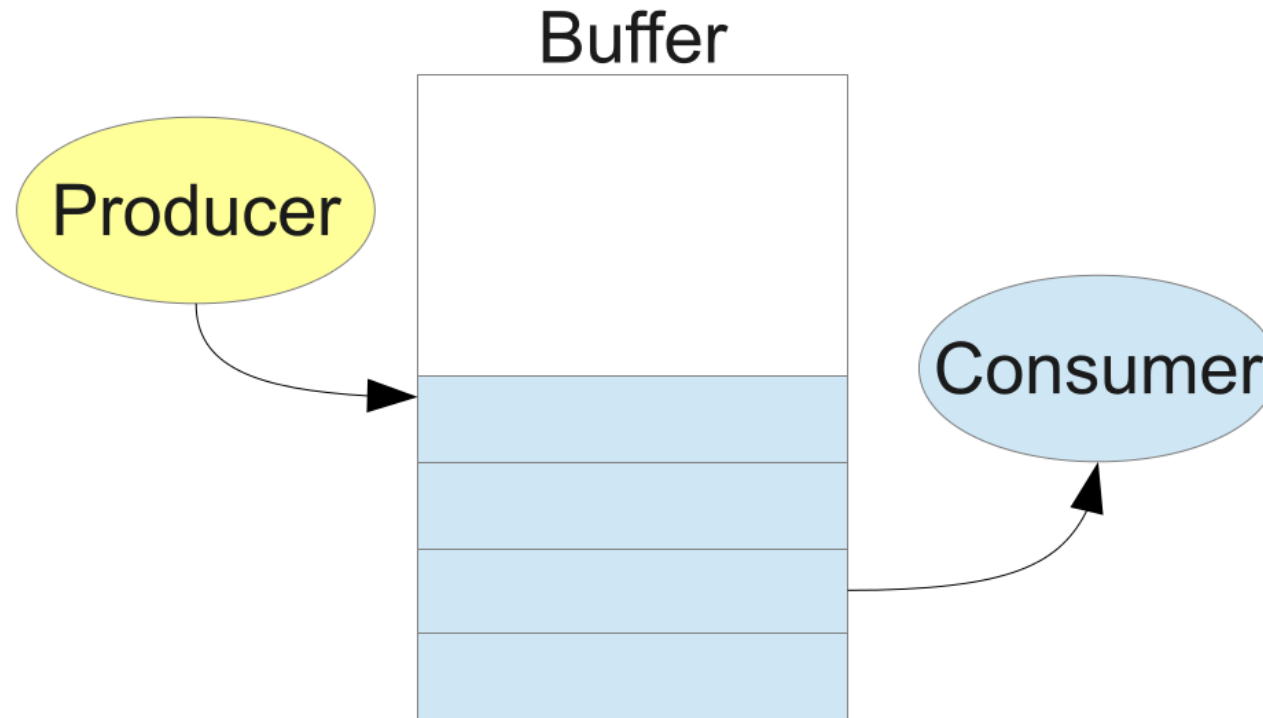


Buffers: BoundedBuffer generico

```
public synchronized void put(Data item)
    throws InterruptedException{
    while(numberInBuffer==size)
        wait();
    last=(last+1)%size;
    numberInBuffer++;
    buffer[last]=item;
    notifyAll();
}
public synchronized Data get()
    throws InterruptedException {
    while(numberInBuffer==0)
        wait();
    first=(first+1)%size;
    numberInBuffer--;
    notifyAll();
    return buffer[first];
}
}
```



Esempio Buffers: Produttore/Consumatore





Esercizio Buffers: Produttore/Consumatore di Frutta

- Il prezzo E varia in funzione del peso P : $E = (P * P)/2$
- Il peso può assumere un valore nell'intervallo $[0.1, 2.5]$
- Alla fine il Produttore stampa il ricavo e il Consumatore stampa i kg di frutta comprati
 - ▶ Il Consumatore scrive NomeThread: Comprati ...Kg di frutta
 - ▶ Il Produttore scrive NomeThread: Ricavato ... euro
- Creare 3 Produttore e 3 Consumatore



Esercizio Buffers: Produttore/Consumatore di Frutta

```
public class Frutta {  
    private double qty;  
    Frutta(double q) {  
        this.qty=q;  
    }  
    public double getPrezzo() {  
        return (this.qty*this.qty)/2;  
    }  
    public double getPeso() {  
        return this.qty;  
    }  
}
```



Esercizio Buffers: Produttore/Consumatore di Frutta

```
public class Consumatore extends Thread {  
    BoundedBuffer<Frutta> mercato;  
    double pesoTotale=0;  
    private String myName;  
    public Consumatore(String name,  
                        BoundedBuffer<Frutta> mercato) {  
        this.setName(name);  
        this.mercato=mercato;  
        this.myName=getName();  
    }  
}
```



Esercizio Buffers: Produttore/Consumatore di Frutta

```
public void run() {  
    for(int i=0; i<20; ++i) {  
        try {  
            Frutta v=mercato.get();  
            pesoTotale+=v.getPeso();  
            System.out.printf("%s: Comprati %.2f Kg\n",  
                               myName, v.getPeso());  
        } catch (InterruptedException e) {}  
    }  
    System.out.printf("%s: Comprati in totale %.2f \n",  
                      myName, pesoTotale);  
}  
}
```



Esercizio Buffers: Produttore/Consumatore di Frutta

```
public class Produttore extends Thread {  
    BoundedBuffer<Frutta> mercato;  
    double totaleSoldi=0;  
    private String myName;  
    public Produttore(String name,  
                        BoundedBuffer<Frutta> mercato) {  
        this.setName(name);  
        this.mercato=mercato;  
        this.myName=getName();  
    }  
}
```



Esercizio Buffers: Produttore/Consumatore di Frutta

```
public void run() {
    for(int i=0; i<20; ++i) {
        try {
            Frutta v=new Frutta(Math.random()*2.4+0.1);
            System.out.printf("%s: vende %.2f Kg\n",
                              myName, v.getPeso());
            mercato.put(v);
            totaleSoldi+=v.getPrezzo();
            System.out.printf("%s: guadagnato %.2f euro\n",
                              myName, v.getPrezzo());
        } catch (InterruptedException e) {}
    }
    System.out.printf("%s: guadagnato %.2f euro in
Totale\n", myName, totaleSoldi);
}
}
```



Esercizio Buffers: Produttore/Consumatore di Frutta

```
public class ProdConsFrutta {  
    public static void main(String[] args) {  
        BoundedBuffer<Frutta> mercato=  
            new BoundedBuffer<Frutta>(8);  
        for(int i=0; i<3; i++) {  
            (new Produttore("P"+i, mercato)).start();  
            (new Consumatore("C"+i, mercato)).start();  
        }  
    }  
}
```



Osservazione

- In realtà non è un bell'esempio: nella realtà la compravendita è una transazione.
- L'esempio visto regge se interpretiamo
 - ▶ Il produttore è colui che produce la frutta e incassa nel momento in cui la porta al mercato
 - ▶ Il consumatore è colui che compra dal mercato.
 - ▶ Il mercato fa da intermediario. Di fatto compra dal produttore e vende al consumatore.
- Altrimenti, il produttore dovrebbe incassare solo nel momento in cui il consumatore preleva la merce dal buffer.



Blackboard

- Una comunicazione di tipo Blackboard è simile a quella di tipo Buffer, **se non che:**
 - ▶ Ha una lettura **non distruttiva** dei dati
 - ▶ I messaggi lasciati sulla Blackboard vengono preservati fino a quanto non vengono sovrascritti, cancellati o invalidati
 - ▶ La scrittura generalmente non è bloccante.
- I messaggi vengono scritti su una Blackboard chiamando **write()**.
- I messaggi vengono cancellati chiamando **clear()**.
- Il metodo **read()** blocca il chiamante fino a quando i dati sulla Blackboard rimarranno non validi.
- La funzione **dataAvailable()** indica se la Blackboard ha dei dati disponibili.
 - ▶ Il lettore la può usare per evitare di bloccarsi



Blackboard

```
public class Blackboard<Data> {  
    private Data theMessage;  
    private boolean statusValid;  
    public Blackboard() {  
        statusValid = false ;  
    }  
    public Blackboard(Data initial) {  
        theMessage = initial;  
        statusValid = true;  
    }  
    public synchronized void write(Data message) {  
        theMessage = message;  
        statusValid = true;  
        notifyAll();  
    }  
    public synchronized void clear() {  
        statusValid = false;  
    }  
}
```

Non bloccante.

Il messaggio corrente viene sovrascritto,
che sia stato letto (da tutti) o meno.

Idem.

Il messaggio viene invalidato, che sia
stato letto (da tutti) o meno.



Blackboard

```
public synchronized Data read() throws InterruptedException {  
    while(!statusValid)  
        wait();  
    return theMessage;  
}  
public boolean dataAvailable() {  
    return statusValid ;  
}  
}
```

Bloccante.
Si aspetta un nuovo messaggio valido.

Esempio Blackboard: Canale Digitale Terrestre

- Un canale digitale terrestre propone ogni giorno una sequenza di programmi
- Un utente vuole vedere le informazioni su cosa stanno trasmettendo in qualsiasi momento
- L'utente si sintonizza e legge le informazioni sul programma che stanno trasmettendo in un determinato istante



Rai 3

18:00 - Per un pugno di libri
18:55 - Meteo 3
19:00 - TG3
19:30 - TG Regione
19:51 - TG Regione Meteo
20:00 - Blob
20:10 - : Che tempo che fa
21:30 - Puliamo l'Italia





Esempio Blackboard: Canale Digitale Terrestre

```
class Programma {
    private String nome;
    private String startTime;
    private int durationInMin=0;
    Programma (String nome, String start, int minutes) {
        this.nome=nome;
        this.startTime=start;
        this.durationInMin=minutes;
    }
    public String toString () {
        return nome + " (inizia alle "+ startTime+ ")";
    }
    public int durationProg(){
        return durationInMin;
    }
}
```



Esempio Blackboard: Canale Digitale Terrestre

```
import java.util.concurrent.ThreadLocalRandom;
class Utente extends Thread {
    String name ;
    Blackboard<Programma> blackboard;
    Utente(Blackboard<Programma> b) {
        this.blackboard=b;
        this.name="Utente" + getName();
    }
    public void run(){
        try {
            System.out.println(this.name + ": attendo programma ");
            Programma msg = blackboard.read();
            System.out.println(this.name + ": sto guardando " +
                               msg.toString());
        } catch (InterruptedException e) { }
    }
}
```



Esempio Blackboard: Canale Digitale Terrestre

```
class CanaleTv extends Thread {
    String name;
    Blackboard<Programma> blackboard;
    Programma[] programmazione;
    CanaleTv(Blackboard<Programma> b, String n, Programma[] p) {
        this.blackboard=b; this.name=n;
        this.programmazione=p;
    }
    public void run(){
        for(int i=0; i<programmazione.length; i++) {
            System.out.println(this.name + " trasmette " +
                               programmazione[i]);
            blackboard.write(programmazione[i]);
            try {
                Thread.sleep(programmazione[i].durationProg());
            } catch (InterruptedException e) {}
        }
    }
}
```



Esempio Blackboard: Canale Digitale Terrestre

```
import java.util.concurrent.ThreadLocalRandom;
public class BlackBoardExample {
    public static void main (String args[])
        throws InterruptedException {
        Blackboard<Programma> bbRai3 = new Blackboard<Programma>();
        Programma[] progs=new Programma[5];
        progs[0]=new Programma("Meteo 3", "20:05", 5);
        progs[1]=new Programma("Blob", "20:10", 15);
        progs[2]=new Programma("TG3", "20:25", 40);
        progs[3]=new Programma("Dott. Stranamore", "21:05", 120);
        progs[4]=new Programma("Notturmo", "23:05", 420);
        new CanaleTv(bbRai3, "Rai3", progs).start();
        for(int i=0; i<8; i++) {
            Thread.sleep(ThreadLocalRandom.current().nextInt(30));
            new Utente(bbRai3).start();
        }
    }
}
```



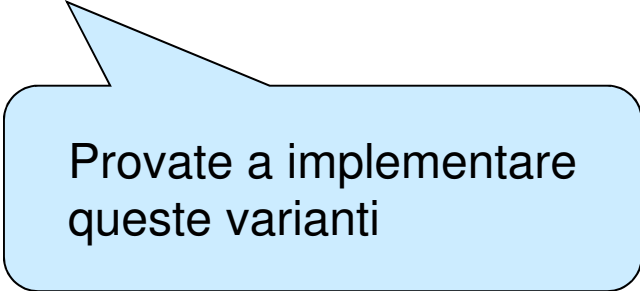
Esempio Blackboard: Canale Digitale Terrestre – un possibile output

```
Rai3 trasmette Meteo 3 (inizia alle 20:05)
Rai3 trasmette Blob (inizia alle 20:10)
UtenteThread-1: attendo un programma
Rai3 trasmette TG3 (inizia alle 20:25)
UtenteThread-1: sto guardando Blob (inizia alle 20:10)
UtenteThread-2: attendo un programma
UtenteThread-2: sto guardando TG3 (inizia alle 20:25)
UtenteThread-3: attendo un programma
UtenteThread-3: sto guardando TG3 (inizia alle 20:25)
Rai3 trasmette Dott. Stranamore (inizia alle 21:05)
UtenteThread-4: attendo un programma
UtenteThread-4: sto guardando Dott. Stranamore (inizia alle 21:05)
UtenteThread-5: attendo un programma
UtenteThread-5: sto guardando Dott. Stranamore (inizia alle 21:05)
UtenteThread-6: attendo un programma
UtenteThread-6: sto guardando Dott. Stranamore (inizia alle 21:05)
UtenteThread-7: attendo un programma
UtenteThread-7: sto guardando Dott. Stranamore (inizia alle 21:05)
UtenteThread-8: attendo un programma
UtenteThread-8: sto guardando Dott. Stranamore (inizia alle 21:05)
Rai3 trasmette Notturmo (inizia alle 23:05)
```




Varianti

- Quali operazioni debbano essere bloccanti può dipendere dalla logica dell'applicazione.
- Ad es.
 - ▶ Potrei non voler sovrascrivere un messaggio che non è ancora stato letto da nessuno.
 - ▶ Potrei non volermi bloccare sulla lettura.
 - ▶ Potrei volermi bloccare in lettura solo per un certo tempo massimo.
 - ▶ ...

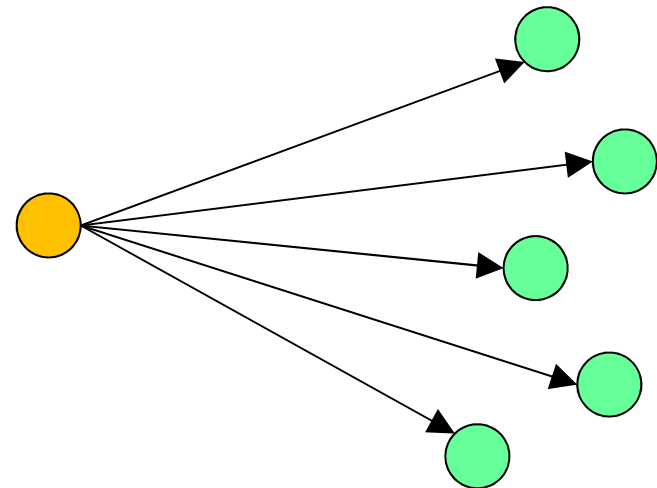


Provate a implementare queste varianti



Broadcast

- Una comunicazione Broadcast permette di inviare dati a più Thread contemporaneamente.
- Solo i thread in attesa ricevono i dati.
- Sono possibili varianti
 - ▶ Se non ci sono Thread in attesa
 - ▶ Se un Broadcast precedente è ancora in corso (cioè non tutti i thread in attesa hanno ricevuto i dati).





Broadcast: esempio classico

- Un trasmettitore radio e un gran numero di ricevitori
- Tutti i ricevitori situati nell'area di copertura del trasmettitore ricevono il segnale
- Il trasmettitore non può sapere con chi ha comunicato





Broadcast

```
public class Broadcast<Data> {  
    private Data theMessage;  
    private boolean arrived;  
    private int waiting;  
    public Broadcast() {  
        arrived = false ;  
        waiting = 0;  
    }  
  
    // per inviare un messaggio in broadcast  
    public synchronized void send(Data message) {  
        if(waiting!=0 && !arrived) {  
            theMessage = message;  
            arrived = true;  
            notifyAll();  
        }  
    }  
}
```

Questa condizione è modificabile a seconda delle esigenze



Broadcast

```
// per mettersi in attesa di un messaggio
public synchronized Data receive() throws InterruptedException {
    try {
        waiting++;
        while(!arrived) // wait for a message to arrive
            wait();
        waiting--;
        if(waiting==0) {
            // The last thread to receive the message resets the flag
            arrived = false ;
        }
    } catch (InterruptedException e) {
        if (--waiting == 0)
            arrived = false;
    }
    return theMessage;
}
```



Un esempio di comunicazione Broadcast: agenzia di stampa

- Una agenzia di stampa, ANSA.it, fornisce le prime notizie su tutto quello che accade in Italia ai giornali che si abbonano al servizio.
- Supponiamo che ci siano 4 giornali interessati a questo servizio: IlCorriere.it, IlFatto.it, LaRepubblica.it e IlMattino.it



Un esempio di comunicazione Broadcast: agenzia di stampa

```
class AnsaMessage extends Thread {  
    Broadcast<String> broadcast;  
    AnsaMessage(Broadcast<String> b) {  
        this.broadcast=b;  
        this.start();  
    }  
    public void run(){  
        String msg;  
        for(int i=0; i<10; i++){  
            try {  
                Thread.sleep(10);  
            } catch (InterruptedException e) { }  
            if(i%2==0) {  
                msg="Aumentano le immatricolazioni";  
            } else {  
                msg="Diminuiscono le immatricolazioni";  
            }  
            System.out.println("News from ANSA.it: "+ msg);  
            broadcast.send(msg);  
        }  
    }  
}
```



Un esempio di comunicazione Broadcast: agenzia di stampa

```
class Newspaper extends Thread {
    String name;
    Broadcast<String> broadcast;
    Newspaper(Broadcast<String> b, String magazineName) {
        this.broadcast=b;
        this.name=magazineName;
    }
    public void run() {
        for(int i=0; i<4; i++){
            System.out.println(this.name + ": attendo news ...");
            try {
                String msg=broadcast.receive();
                System.out.println(this.name+": "+msg);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```




Un esempio di comunicazione Broadcast: agenzia di stampa

```
public class BroadcastExample {  
    static Broadcast<String> broadcast=new Broadcast<String>();  
    public static void main(String args[])  
        throws InterruptedException {  
        new AnsaMessage(broadcast);  
        new Newspaper(broadcast, "IlCorriere.it").start();  
        new Newspaper(broadcast, "IlFatto.it").start();  
        new Newspaper(broadcast, "LaRepubblica.it").start();  
        new Newspaper(broadcast, "IlMattino.it").start();  
    }  
}
```



Un esempio di comunicazione Broadcast: agenzia di stampa – possibile output

```
IlCorriere.it: attendo news ...
IlFatto.it: attendo news ...
IlMattino.it: attendo news ...
LaRepubblica.it: attendo news ...
News from ANSA.it: Aumentano le immatricolazioni
IlCorriere.it: Aumentano le immatricolazioni
IlCorriere.it: attendo news ...
IlMattino.it: Aumentano le immatricolazioni
IlFatto.it: Aumentano le immatricolazioni
IlFatto.it: attendo news ...
LaRepubblica.it: Aumentano le immatricolazioni
LaRepubblica.it: attendo news ...
IlMattino.it: attendo news ...
News from ANSA.it: Diminuiscono le immatricolazioni
IlCorriere.it: Diminuiscono le immatricolazioni
IlFatto.it: Diminuiscono le immatricolazioni
LaRepubblica.it: Diminuiscono le immatricolazioni
LaRepubblica.it: attendo news ...
IlMattino.it: Diminuiscono le immatricolazioni
IlMattino.it: attendo news ...
IlFatto.it: attendo news ...
IlCorriere.it: attendo news ...
```



Esercizio Broadcast: Conferenze

- Durante una conferenza più speakers parlano in diverse stanze
- Gli interessati entrano nella stanza dove si trattano gli argomenti di loro interesse e aspettano lo speaker
- Lo speaker dà il messaggio di benvenuto a tutti i potenziali interessati, poi pronuncia un discorso.
- Il discorso consta di una serie di frasi, intervallate da pause.
- Gli ascoltatori possono alzarsi e andare via prima della fine del discorso.



Esercizio Broadcast: Conferenze

```
public class Conferenza {  
    public static void main(String[] args)  
        throws InterruptedException {  
        Broadcast<String> speechMessage=new Broadcast<String>();  
        System.out.println("La sala conferenze e' aperta...");  
        for(int i=0; i<10; i++) {  
            new Ascoltatore(speechMessage).start();  
        }  
        Thread.sleep(1000);  
        System.out.println("Entra lo speaker...");  
        new Speaker(speechMessage).start();  
    }  
}
```



Esercizio Broadcast: Conferenze

```
public class Ascoltatore extends Thread {
    Broadcast<String> speechMessage;
    public Ascoltatore(Broadcast<String> wm) {
        speechMessage=wm;
        setName("Ascoltatore" + getName());
    }
    public void run() {
        try {
            System.out.println(getName()+" in attesa di ascoltare");
            int numAscolti=(int) (Math.random()*10);
            for(int i=0; i<numAscolti; i++) {
                String msg=speechMessage.receive();
                System.out.println(getName()+" ha sentito: "+msg);
            }
            System.out.println(getName()+" esce");
        } catch (InterruptedException e) {}
    }
}
```



Esercizio Broadcast: Conferenze

```
public class Speaker extends Thread {
    Broadcast<String>speechMessage;
    public Speaker(Broadcast<String> wm) {
        speechMessage=wm;
        setName("Speaker " + getName());
    }
    public void run() {
        System.out.println(getName()+": Inizia a parlare...");
        speechMessage.send("Benvenuti!");
        for(int i=0;i<10;i++) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) { }
            speechMessage.send("Bla bla bla "+i);
        }
    }
}
```



Broadcast: varianti

```
// per inviare un messaggio in broadcast
public synchronized void send(Data message) {
    if(waiting!=0 && !arrived){
        theMessage = message;
        arrived = true;
        notifyAll();
    }
}
```

- In questa implementazione, il messaggio viene inviato solo se:
 - ▶ Ci sono ascoltatori (waiting non nullo)
 - ▶ Il messaggio precedente è stato ricevuto da tutti
- Non è detto che debba essere sempre così: entrambe le condizioni potrebbero essere rilassate.
- Dipende dal comportamento che si vuole ottenere!



Barrier

- Una barriera blocca un insieme di thread fino a quando tutti non raggiungono la barriera.
- I Thread che desiderano bloccarsi ad una barriera devono chiamare il metodo **waitB**.
- Se la barriera deve ancora rilasciare i Thread, allora tutti i Thread che arrivano vengono trattenuti.
- Quando l'ultimo Thread arriva, tutti i Thread vengono rilasciati.
- Non c'è comunicazione, solo sincronizzazione. Se si desidera comunicare dati, va fatto a parte.



Barrier

```
public class Barrier {
    private int needed;
    private int arrived;
    private boolean releasing;
    public Barrier(int number) {
        needed=number; // Number of threads to block
        arrived=0;
        releasing=false;
    }
    public synchronized int value () {
        return arrived ;
    }
    public int capacity() {
        return needed;
    }
}
```



Barrier

```
public synchronized void waitB() throws InterruptedException {
    while(releasing)
        wait(); // bloccati alla barriera
    try {
        arrived++;
        while(arrived<needed && !releasing)
            wait();
        if(arrived==needed) { // When all present
            releasing=true;
            System.out.println("All "+needed+" parties reached barrier");
            notifyAll();
        }
    } finally {
        arrived--;
        if(arrived==0) { // l'ultimo thread che lascia la barriera
            releasing=false;
            notifyAll(); // sveglia i nuovi thread
        }
    }
}
```

Arrivato quando la barriera è aperta (troppo presto): aspetta.



Un esempio con 3 Thread che usano una Barriera

```
import java.util.concurrent.*;
// Runnable task for each thread
public class Task implements Runnable {
    private Barrier barrier;
    public Task(Barrier barrier) {
        this.barrier=barrier;
    }
    public void run() {
        try {
            for(int i=0; i<4; i++){
                Thread.sleep(ThreadLocalRandom.current().nextInt(30));
                System.out.println(Thread.currentThread().getName()+
                                   " arrived at the barrier");

                barrier.waitB();
                System.out.println(Thread.currentThread().getName()+
                                   " has crossed the barrier");
            }
        } catch (InterruptedException ex) {}
    }
}
```



Un esempio con 3 Thread che usano una Barriera

```
public class BarrierExample {  
    public static void main(String args[]) {  
        // creating a Barrier with 3 parties  
        // i.e. 3 Threads need to call waitB()  
        final Barrier cb=new Barrier(3);  
        // starting each thread  
        Thread t1=new Thread(new Task(cb), "Thread 1");  
        Thread t2=new Thread(new Task(cb), "Thread 2");  
        Thread t3=new Thread(new Task(cb), "Thread 3");  
        t1.start();  
        t2.start();  
        t3.start();  
    }  
}
```



Un esempio con 3 Thread che usano una Barriera – possibile output

```
Thread 1 arrived at the barrier
Thread 2 arrived at the barrier
Thread 3 arrived at the barrier
All 3 parties arrived at barrier
Thread 3 has crossed the barrier
Thread 1 has crossed the barrier
Thread 2 has crossed the barrier
Thread 1 arrived at the barrier
Thread 3 arrived at the barrier
Thread 2 arrived at the barrier
All 3 parties arrived at barrier
Thread 2 has crossed the barrier
Thread 1 has crossed the barrier
Thread 3 has crossed the barrier
Thread 2 arrived at the barrier
Thread 1 arrived at the barrier
Thread 3 arrived at the barrier
All 3 parties arrived at barrier
Thread 3 has crossed the barrier
Thread 2 has crossed the barrier
Thread 1 has crossed the barrier
Thread 2 arrived at the barrier
```



Esempio Barrier : Corsa di Cavalli

- Durante una corsa di cavalli, ogni cavallo deve entrare nella barriera prima di partire
- Esiste una postazione per ogni cavallo
- Ad ogni giro i cavalli devono fermarsi alla barriera e aspettare tutti gli altri
- Ad ogni giro un cronista annuncia il numero di giri fatti
- Ci sono N cavalli/postazioni e bisogna fare M giri



Esempio Barrier: Corsa di Cavalli

```
public class CorsaDiCavalli {
    private static final int NUM_CAVALLI=3;
    private static final int NUM_GIRI=2;
    public static void main(String args[])
        throws InterruptedException {
        Barrier barriera=new Barrier(NUM_CAVALLI);
        for(int i=0; i<NUM_CAVALLI; i++) {
            new Cavallo(barriera, NUM_GIRI).start();
        }
        new Cronista(barriera, NUM_GIRI).start();
    }
}
```



Esempio Barrier : Corsa di Cavalli

```
public class Cronista extends Thread {
    Barrier barriera ;
    private int numGiri ;
    public Cronista(Barrier b, int n) {
        barriera=b;
        setName("Cronista");
        numGiri=n;
    }
    public void run() {
        for(int i=0; i<numGiri; i++) {
            try {
                int c=barriera.waitCycle(i+1);
                System.out.println(this.getName()+
                                   ": completato il giro num."+c);
            } catch(InterruptedException e) {}
        }
        System.out.println(this.getName()+": ultimo giro!");
    }
}
```




Esempio Barrier : Corsa di Cavalli

```
public class Cavallo extends Thread {
    private int numGiri;
    Barrier barriera;
    public Cavallo(int id, Barrier b, int n) {
        barriera=b;
        setName("Cavallo " + id);
        numGiri=n;
    }
    public void run() {
        System.out.println(getName()+ " e` partito!");
        for(int i=0; i<numGiri; i++) {
            try {
                Thread.sleep((long) (Math.random()*3000));
                System.out.println(getName()+" entra nella barriera ");
                barriera.waitB();
                System.out.println(getName()+ " e` ripartito!");
            } catch (InterruptedException ex ) {}
        }
        System.out.println(getName()+ " termina la corsa!");
    }
}
```



Esempio Barrier : Corsa di Cavalli

```
public class Barrier {
    private int needed, arrived, numCycles;
    private boolean releasing;
    public Barrier(int number) {
        needed=number; // Number of threads to block
        arrived=0; numCycles=0;
        releasing=false;
    }
    public synchronized int value () {
        return arrived ;
    }
    public synchronized int waitCycle(int cycleNum)
                                   throws InterruptedException {
        if(cycleNum!=numCycles)
            wait();
        return numCycles;
    }
    public int capacity() {
        return needed;
    }
}
```



Esempio Barrier : Corsa di Cavalli

```
public synchronized void waitB() throws InterruptedException {
    while(releasing)
        wait(); // Wait for previous batch to depart
    try {
        arrived++;
        while(arrived!=needed && !releasing)
            wait();
        if(arrived==needed) { // When all present
            releasing=true;
            System.out.println("All "+needed+
                               " parties arrived at barrier");

            numCycles++;
            notifyAll();
        }
    } finally {
        arrived--;
        if(arrived==0) { // Last thread to leave
            releasing=false;
            notifyAll(); // Allow new batch to arrive
        }
    }
}
```



Esercizio Barrier: Corsa di Cavalli

Cavallo 1 è partito!
Cavallo 2 è partito!
Cavallo 3 è partito!
Cavallo 2 sta per entrare nella barriera
Cavallo 1 sta per entrare nella barriera
Cavallo 3 sta per entrare nella barriera
All 3 parties arrived at barrier
Cavallo 3 è ripartito!
Cavallo 1 è ripartito!
Cavallo 2 è ripartito!
Cronista: completato il giro num.1
Cavallo 1 sta per entrare nella barriera
Cavallo 3 sta per entrare nella barriera
Cavallo 2 sta per entrare nella barriera
All 3 parties arrived at barrier
Cavallo 2 è ripartito!
Cavallo 3 è ripartito!
Cronista: completato il giro num.2
Cavallo 1 è ripartito!

Cavallo 3 sta per entrare nella barriera
Cavallo 2 sta per entrare nella barriera
Cavallo 1 sta per entrare nella barriera
All 3 parties arrived at barrier
Cavallo 1 è ripartito!
Cavallo 2 è ripartito!
Cavallo 2 termina la corsa!
Cronista: completato il giro num.3
Cavallo 3 è ripartito!
Cavallo 1 termina la corsa!
Cavallo 3 termina la corsa!
Cronista: ultimo giro!



Esercizio Barrier: Corsa di Cavalli

- Provando a eseguire, si vede che il cronista è spesso in ritardo.
- Possiamo evitare questo effetto usando le librerie di Java



CyclicBarrier Java

- CyclicBarrier è un sincronizzatore introdotto nel JDK 5 nel package `java.util.concurrent`.



java.util.concurrent.CyclicBarrier

```
public class CyclicBarrier  
    extends Object
```

A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called cyclic because it can be re-used after the waiting threads are released.

A CyclicBarrier supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This barrier action is useful for updating shared-state before any of the parties continue.



java.util.concurrent.CyclicBarrier: Constructors

CyclicBarrier(int parties)

Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and does not perform a predefined action when the barrier is tripped.

CyclicBarrier(int parties, Runnable barrierAction)

Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and which will execute the given barrier action when the barrier is tripped, performed by the last thread entering the barrier.



java.util.concurrent.CyclicBarrier: Methods

int await()

Waits until all parties have invoked **await** on this barrier.

int await(long timeout, TimeUnit unit)

Waits until all parties have invoked **await** on this barrier, or the specified waiting time elapses.

int getNumberWaiting()

Returns the number of parties currently waiting at the barrier.

int getParties()

Returns the number of parties required to trip this barrier.

boolean isBroken()

Queries if this barrier is in a broken state.

void reset()

Resets the barrier to its initial state.



Riscriviamo la corsa di cavalli

- Usando CyclicBarrier



CorsaDiCavalli

```
import java.util.concurrent.CyclicBarrier;

public class CorsaDiCavalli {
    private static final int NUM_CAVALLI=3;
    private static final int NUM_GIRI=3;
    public static int num_giri=1;
    public static void main(String args[])
        throws InterruptedException {
        CyclicBarrier barriera=new CyclicBarrier(NUM_CAVALLI,
                                                    new Cronista());

        for(int i=0; i<NUM_CAVALLI; i++) {
            new Cavallo(barriera, NUM_GIRI).start();
        }
    }
}
```



Cavallo

```
import java.util.concurrent.*;
public class Cavallo extends Thread {
    private int numGiri;
    CyclicBarrier barriera;
    public Cavallo(int id, CyclicBarrier b, int n) {
        barriera=b; numGiri=n;
        setName("Cavallo_" + id);
    }
    public void run() {
        for(int i=0; i<numGiri; i++) {
            try {
                Thread.sleep(ThreadLocalRandom.current().nextInt(100, 300));
                System.out.println(getName()+" entra nella barriera ");
                barriera.await();
                System.out.println(getName()+ " e` ripartito!");
            } catch (Exception e) {}
        }
        System.out.println(getName()+" termina");
    }
}
```



Cronista

```
public class Cronista implements Runnable {  
    public void run() {  
        System.out.println("I cavalli partono per il giro "+  
                           CorsaDiCavalli.num_giri++);  
    }  
}
```



Nuovo possibile output

```
Cavallo_2 sta per entrare nella barriera
Cavallo_1 sta per entrare nella barriera
Cavallo_3 sta per entrare nella barriera
I cavalli partono per il giro 1
Cavallo_3 e` ripartito!
Cavallo_2 e` ripartito!
Cavallo_1 e` ripartito!
Cavallo_3 sta per entrare nella barriera
Cavallo_2 sta per entrare nella barriera
Cavallo_1 sta per entrare nella barriera
I cavalli partono per il giro 2
Cavallo_1 e` ripartito!
Cavallo_3 e` ripartito!
Cavallo_2 e` ripartito!
Cavallo_1 sta per entrare nella barriera
Cavallo_2 sta per entrare nella barriera
Cavallo_3 sta per entrare nella barriera
I cavalli partono per il giro 3
Cavallo_3 e` ripartito!
Cavallo_1 e` ripartito!
Cavallo_2 e` ripartito!
Cavallo_2 termina      Cavallo_3 termina      Cavallo_1 termina
```



Un esempio di uso delle CyclicBarrier Java: somma righe matrice

- Sia A una matrice di dimensione $n \times m$.
- Si vuole ottenere il vettore C i cui elementi sono definiti come segue:

$$c_i = \sum_{k=0}^{m-1} a_{ik}$$

dove $0 \leq i < n$

- Implementare un programma concorrente per calcolare gli elementi c_i in parallelo.
 - ▶ un thread si occupa di c_0 ,
 - ▶ un altro che si occupa di c_1 ,
 - ▶ uno per c_2 ,
 - ▶ ecc.
- Alla fine dei calcoli, viene visualizzato C
 - ▶ Usiamo CyclicBarrier per eseguire la parte finale dopo aver completato i singoli thread



RowSummer

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

public class RowSummer extends Thread {
    private int row; // indice della riga su cui lavorare
    private int matrix[][];
    private int results[];
    private CyclicBarrier myBarrier;
    public RowSummer(int row, int m[][], int r[],
                     CyclicBarrier b) {
        this.row=row;
        this.matrix=m;
        this.results=r;
        this.myBarrier=b;
    }
}
```




RowSummer

```
public void run() {  
    int columns=matrix[row].length;    // numero di colonne  
    int sum=0;  
    for(int i=0; i<columns; i++) {  
        sum += matrix[row][i];  
    }  
    results[row]=sum;  
    System.out.println("Result for row "+row+" is: " + sum);  
    try {  
        myBarrier.await();  
    } catch (InterruptedException | BrokenBarrierException e){}  
}  
}
```



ParMatSum (main)

```
import java.util.concurrent.*;
public class ParMatSum {
    private static int matrix[][] =
        { { 1, 2, 3, 4, 5 } ,
          { 2, 2, 2, 2, 2 } ,
          { 3, 3, 3, 3, 3 } ,
          { 4, 4, 4, 4, 3 } ,
          { 5, 5, 5, 5, 5 } } ;
    public static int results[];
    public static void main(String args[]) throws InterruptedException {
        final int rows=matrix.length;    // number of rows
        final int cols=matrix[0].length; // number of columns
        for(int i=0; i<rows; i++){
            System.out.print("[");
            for(int j=0; j<cols; j++){ System.out.print(matrix[i][j]+" "); }
            System.out.println("]");
        }
        results=new int[rows];
        CyclicBarrier bar=new CyclicBarrier(rows, new ResultUser(results));
        for(int i=0; i<rows; i++) {
            new RowSummer(i, matrix, results, bar).start();
        }
    }
}
```



ResultUser

```
public class ResultUser implements Runnable{
    private int results[];
    public ResultUser(int r[]){
        this.results=r;
    }
    public void run() {
        int total=0;
        System.out.print("[");
        for(int i=0; i<results.length; i++){
            total+=results[i];
            System.out.print(results[i]+" ");
        }
        System.out.println("]");
        System.out.print("total = "+total);
    }
}
```



Output

```
Result for row 0 is: 15
Result for row 1 is: 10
Result for row 2 is: 15
Result for row 4 is: 25
Result for row 3 is: 19
[15 10 15 19 25 ]
total = 84
```

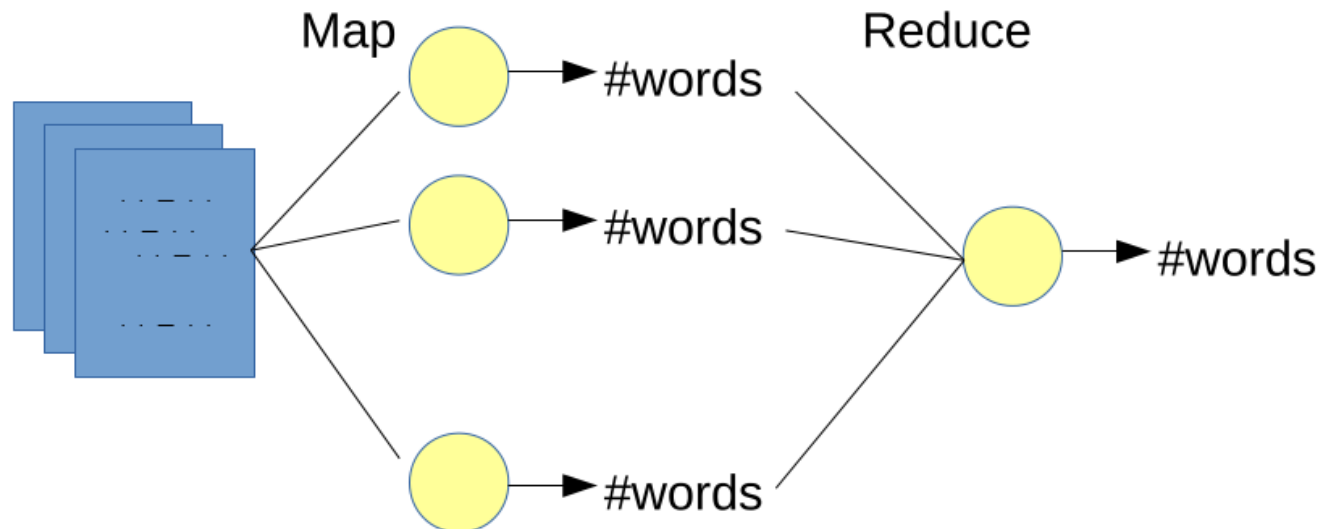


Map-Reduce usando CyclicBarrier

- MapReduce è un framework software definito da Google per supportare la computazione di grandi quantità di dati in cluster di computer.
- Proviamo a realizzare il concetto di MapReduce con la programmazione concorrente e usando gli strumenti che abbiamo.

Map-Reduce usando CyclicBarrier

- Vogliamo usare CyclicBarrier per dividere un compito grande in un insieme di piccoli task, esattamente come fa Map-Reduce.
- Vediamo come funziona MapReduce tramite un semplice esempio WordCount: contare le parole appartenenti ad un insieme di file di testo.





Map-Reduce usando CyclicBarrier

- Prendiamo tutte le descrizioni delle borse presenti nel sito <http://www.misshobby.com/it/categoria/donna/borse>
- Ogni descrizione sta in un file, che ha un contenuto di questo tipo:

Borsa in fettuccia, foderata (fodera chiara a righe azzurre) e con due taschini interni. Fornita con manici rigidi e tracolla sganciabile. Chiusura con bottone.

Realizzata interamente a mano, cuciture comprese.

Dimensioni 32×18×8, peso 550g
- Contiamo tutte le parole in totale suddividendo il lavoro in sotto-task
 - ▶ Ogni task si occupa di un insieme di file



Map-Reduce usando CyclicBarrier

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

public class WordCounter implements Runnable {
    CyclicBarrier cyclicBarrier;
    int wordCount=0;
    int startFile, endFile;
    WordCounter(CyclicBarrier c, int startFile, int endFile) {
        cyclicBarrier=c;
        this.startFile=startFile;
        this.endFile=endFile;
        new Thread(this).start();
    }
}
```




Map-Reduce usando CyclicBarrier

```
public void run() {  
    BufferedReader br = null;  
    try {  
        for(int i=startFile; i<endFile; i++) {  
            br=null;  
            String fileName="file_" + i + ".txt";  
            System.out.println("leggo "+fileName);  
            br=new BufferedReader(new FileReader("../"+fileName));  
            String line=br.readLine();  
            while(line!=null) {  
                String[] wordArray=line.split("\\ s+");  
                wordCount+=wordArray.length;  
                line=br.readLine();  
            }  
        }  
        System.out.println("GOING TO WAIT");  
        cyclicBarrier.await();  
    } catch(Exception exc) { System.out.println(exc); }  
    try { br.close(); } catch (IOException e) {}  
}}
```



Map-Reduce usando CyclicBarrier

```
public class BarrierReachedAction implements Runnable {
    private ArrayList<WordCounter> wordCounters;
    long t0=0;
    public BarrierReachedAction(long start) {
        t0=start;
    }
    public void setCounters(ArrayList<WordCounter> wordCounters) {
        this.wordCounters=wordCounters;
    }
    public void run() {
        System.out.println("BarrierReached!");
        int totalWords=0;
        for(WordCounter wc:wordCounters) {
            totalWords+=wc.wordCount;
        }
        long totalTimeElapsed=System.currentTimeMillis()-t0;
        System.out.println("Elapsed time = "+totalTimeElapsed);
        System.out.println("Word count is = " + totalWords);
    }
}
```



Map-Reduce usando CyclicBarrier

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

public class WordCountExample {
    public static void main(String args[]) {
        ArrayList<WordCounter> counters=new ArrayList<WordCounter>();
        long t0=System.currentTimeMillis();
        BarrierReachedAction reduce=new BarrierReachedAction(t0);
        int parties=8;
        CyclicBarrier cyclicBarrier=new CyclicBarrier(parties,
                                                    reduce);

        int incr=400/parties;
        for(int i=0; i<parties; i++) {
            counters.add(new WordCounter(cyclicBarrier,
                                         (i)*incr, (i+1)*incr));
        }
        reduce.setCounters(counters);
    }
}
```



Map-Reduce usando CyclicBarrier: output

```
leggo file_0.txt
leggo file_100.txt
leggo file_50.txt
. . .
leggo file_149.txt
leggo file_195.txt
GOING TO WAIT
GOING TO WAIT
leggo file_298.txt
leggo file_196.txt
leggo file_299.txt
leggo file_197.txt
GOING TO WAIT
leggo file_198.txt
leggo file_199.txt
GOING TO WAIT
BarrierReached!
Elapsed time = 80
Word count is = 3179
```



Conclusioni

- Nella programmazione concorrente, esistono molti modi per far comunicare i Thread e diversi paradigmi di sincronizzazione.
- Abbiamo visto alcuni dei meccanismi più comunemente usati.
- NB: ce ne sono comunque diversi altri, e chiunque può inventarsi delle varianti che meglio si adattino al caso in questione.