A complete example

The Fontebella baths

Semaphores, Regions, Monitors, Ada-Java

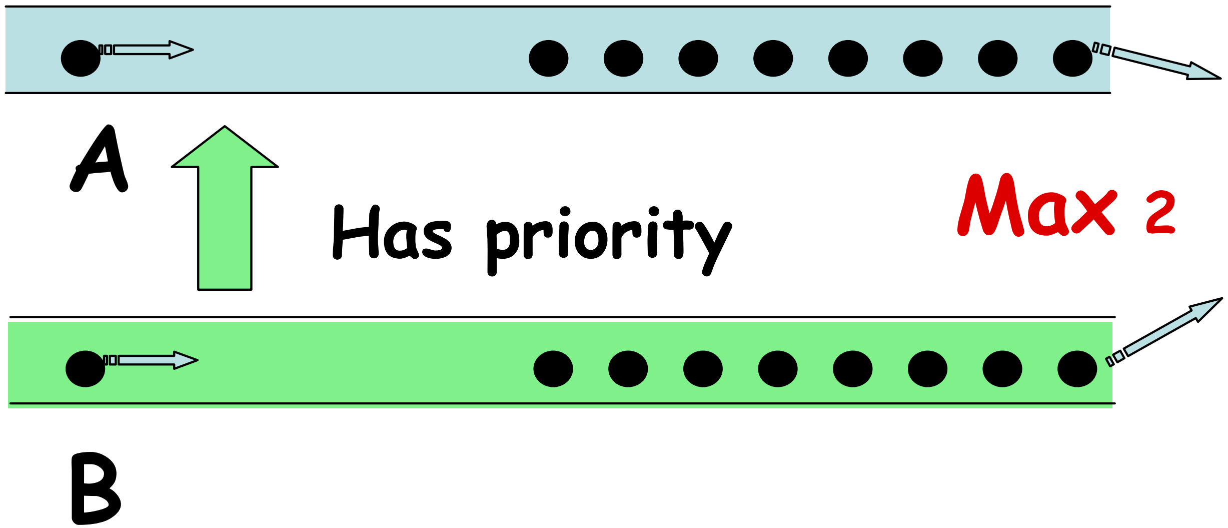
The Fontebella baths - 1

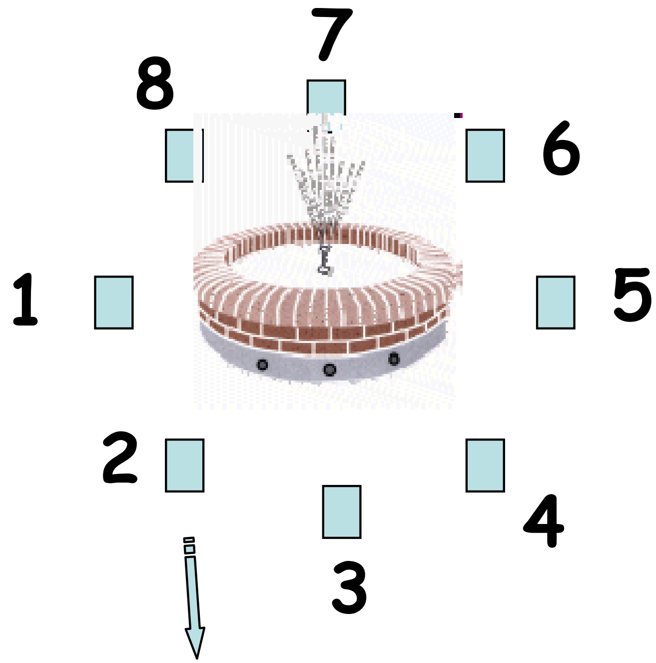
In the Fontebella baths there is a fountain with **8 spouts**, customers can go there to fill a mug.
A spout fills a mug in 15.5 s.
There are 2 waiting queues, **A for normal customers and B for those with special diseases**.
The peaceful environment permits a self-discipline among customers who access the fountain respecting the priority.

1. The customer at the head of one queue, when **the other queue is empty**, accesses the fountain as soon as a spout is available;
2. Otherwise each customer in the A queue **gives way** to at most two customers in the B queue so that these last have priority, then she can access the fountain

(these are **two** if, when one in A should give way to second one in B, there is actually a customer in B, otherwise only **one** in B gets priority).

The Fontebella baths - 2





The Fontebella baths - 3

Design the **(self-)control system** of the two queues using each of the presented techniques:

- **Semaphores,**
- **Critical Regions,**
- **Hoare's Monitor and**
- **Java Monitor**
- **Ada-Java**

And represent the **customers** with threads which are generated with one of the types (A or B) e with a random frequency between 0.5 and 2 s.

Each **thread** executes the sequence:

- enters the queue calling, according to its type, **entraCodaA** or **entraCodaB**, in general suspensive, which returns the index 1..8 of the assigned spout),
- fills the mug,
- exit the area calling the method **fineRiempimento**.

Analysis

Read carefully the text, recognizing the important requirements that must be thoroughly respected

1. Define **shared variables** (buffers, 'pointers', state and counting variables, etc.)
2. Identify **synchronization conditions**
3. Insert required synchronizations within the **requested methods** in order to fulfill the requirements

If you define **threads as inner classes** in the application class, **shared variables** and synchronization methods, defined as (even private) elements of the containing class, are **accessible** to the inner class methods.

.

Shared variable

- **Counting** the number of **customers** in each queue
- A **stat** counter, inizialized to **2**, representing for each A customer how many B customers to give priority; it must be reset to **2** at any time a A customer goes to the fountain
- **Counting** the customers at the fountain
- An index representing the **last occupied spout** (observe that, having the filling a fixed duration, spouts are occupied and freed circularly)
- Other variables depending on the used synchronization tool

Synchronization conditions

- **a A customer must wait**
  - If she is not at the head of the A queue
  - If there is no free spout
  - If a B customer has priority
    (some B customers are waiting and the **stat** counter is NOT equal to 0)

Semaphores
Monitor (BH, MdH, NdJ)

- **a B customer must wait**
  - If she is not at the head of the B queue
  - If there is no free spout
  - If a A customer has priority
    (some A customers are waiting and the **stat** counter is EQUAL to 0)

Synchronized methods

We call the application class **Fontana*Type*** with *Type*=**(Sem,Reg,Mon,Jav)** according to the synchronization tool. It includes these synchronized methods:

1. **int entraCodaA()** where a A customer may be forced to wait
2. **int entraCodaB()** where a B customer may be forced to wait
3. **fineRiempimento()** which permits a waiting customer, if present, to enter in compliance with the synchronization rules

Thread - 1

- The two types of customers are respectively represented by **ClienteATh** e **ClienteBTh** extending the **Thread** class (they do not have to extend another class)

- The two classes are defined as **inner member classes** of **Fontana** so that they can access the shared variables in the associated instance of **Fontana** (notice that these shared variables are private)

Thread - 2

- The code in their **run()** method executes the sequence described in the text;
  between **entraCodax()** e **fineRiempimento()** it must elapse 15.5 s for the filling
- The main thread (the one executing the **main** method) is in charge of creating the client threads

Development phases

1. **Petri net** optional
2. **main() method**
   - Creation of instances
   - Thread activations
3. **Thread classes**
   - constructor
   - run() method
4. **Synchronization class**
   - Synchronization tools

Main (FontanaXXX)


```
public static void main(String[] args) {
    System.err.println("** Battere Ctrl-C per terminare!");
    FontanaTipo fo = new FontanaTipo();
    int cnt=1;
    for(;;) {
        Util.rsleep(500, 2000);
        if (Util.randVal(1,2) == 1)
            fo.new ClienteATh ("num"+(cnt++)).start();
        else
            fo.new ClienteBTh ("num"+(cnt++)).start();
    }
} //[m][s] main
```

XXX = [Sem,Reg,Mon,Jav]

FontanaSem,
FontanaReg,
FontanaMon,
FontanaJav

Main (FontanaXXX)


```
public static void main(String[] args) {
    System.err.println("** Battere Ctrl-C per terminare!");
    FontanaXXX fo = new FontanaXXX ();
    int cnt=1;
    for(;;) {
        Util.rsleep(500, 2000);
        if (Util.randVal(1,2) == 1)
            fo.new ClienteATh ("num"+(cnt++)).start();
        else
            fo.new ClienteBTh ("num"+(cnt++)).start();
    }
} //[m][s] main
```

XXX = [Sem,Reg,Mon,Jav]

FontanaSem,
FontanaReg,
FontanaMon,
FontanaJav

thread client


```
private class ClienteATh extends Thread {

    public ClienteATh(String name) {
        super(name);   }

    public void run() {
        System.out.println("!!! Il cliente "+
            getName()+" di tipo A va in coda");
        int zamp=entraCodaA();         // arriva e attende
        System.out.println("+++ Il cliente "+
            getName()+" di tipo A va a bere allo zampillo "+zamp);
        Util.sleep(RIEMPIMENTO);  // beve
        System.out.println("--- Il cliente "+
            getName()+" di tipo A lascia lo zampillo "+zamp);
        fineRiempimento();                 // lascia la fontana
        } //[m] run
    } // {c} ClienteATh
```

Similar for client B

thread client (simplified)


```
private class ClienteATh extends Thread {
  public ClienteATh(String name) {
    super(name);  }

  public void run() {
    int zamp=entraCodaA();      // arriva e attende
    Util.sleep(RIEMPIMENTO); // riempie il boccale
    fineRiempimento();            // lascia la fontana
    } //[m] run

} // {c} ClienteATh
```

Similar for client B

Solution with Semaphores - 1

- Evaluate if it is possible to map some synchronization onto single (bianry or counting)

- When the synchronization **complexity** is high, **adopt** the '**private sempahore**' approach

Solution with Semaphores - 2

- Notice that if you declare
  Semaphore priv1 = new Semaphore();
  the semaphore is actually **private** of the creating thread;
  when you declare:
  Semaphore priv2 = new Semaphore(false);
  this is a generic binary semaphore but you can 'consider' it as a semaphore private to a thread or to a subset of threads
  (without any run-time control)
- For usually the solution requires more than one semaphores, the application class **does not extend the Semaphore class** but it includes as attributes a certain number of **istances** of this class.

Semaphores: variables


```
public class FontanaSem {
   private static final long Riempimento = 15500L;
      // tempo di riempimento
   private int zampilliLiberi = 8;
      // i clienti in fontana saranno 8-zampilliLiberi
   private int ultimoZampillo = 7;
      // ultimo zampillo occupato (0..7)
   private Semaphore mutex = new Semaphore(true);
      // protezione della sezione critica
   private Semaphore attesaA = new Semaphore(false);
   private Semaphore attesaB = new Semaphore(false);
      // semafori privati dei clienti in attesa
      // i clienti in coda si ottengono dal contatore del semaforo
   private int stat = 2;
      // conteggio per priorità` clienti B
```

Semaphores: Synchronization methods

1. **int** **entraCodaA()** includes the waiting for a A client
2. **int** **entraCodaB()** includes the waiting for a B client
3. **fineRiempimento()** which permits a waiting customer, if present, to enter in compliance with the synchronization rules

recall

Semaphore: entraCodaA (semplified)


```
public int entraCodaA() {
  mutex.p();            // entra in mutua esclusione
  // verifica la condizione di attesa sul sem. privato A
  if (attesaA.queue() > 0 || zampilliLiberi == 0 ||
       (attesaB.queue()>0 && stat != 0)) {
    // deve attendere
    mutex.v();
    attesaA.p();        // risvegliato in mutua esclusione
    }
  zampilliLiberi--;  // assegna zampillo in mutex
  stat=2;               // reset di stat
  int zamp = (ultimoZampillo = (ultimoZampillo+1)%8)+1;
  mutex.v();
  return zamp;
} //[m] entraCodaA
```

Condition for A

Resumed with mutex

Only now exit from mutex

race condition with private semaphore - 1


```
mx.p(); ///----------\\
//valutazione di cond_sincr
if (forks[i] && forks[(i+1)%N]) {
   // aggiorna variabili
   forks[i] = false;
   forks[(i+1)%N]] = false;
   priv[i].v();}
else
   waiting[i] = true;
mx.v();  ///---------//
priv[i].p();
//
//
mx.p();  ///---------\\
waiting[i] = false;
forks[i] = false;
forks[(i+1)%N]] = false;
mx.v();  ///---------//
// mangia
...
```

Qui può 'intrufolarsi' il filosofo adiacente

```
mx.p();  // entra in sezione critica
forks[i] = true;
forks[(i+1)%N] = true;
//valutazione di cond_sincr per gli adiacenti
if (waiting[(i-1+N)%N] && forks[(i-1+N)%N])
    priv[(i-1+N)%N].v();
if (waiting[(i+1)%N] && forks[(i+2)%N])
    priv[(i+1)%N].v();
mx.v(); //esce da sezione critica
```

recall

race condition with private semaphore - 2

recall

mx.p();   // entra in sezione critica
if (! <condizione>) {
  mx.v(); // rilascia la sez. critica
  priv.p(); //<attende>
  ~~mx.p();~~
}
<aggiorna variabili>
mx.v();   // esce da sezione critica

It resumes already in mutual exclusion

mx.p();   // entra in sezione critica
<aggiorna variabili>
if (<condizione per altro> && <in attesa>)
    priv.v();
else
    mx.v();   // esce da sezione critica

It wakes up the other and yields mutex

Semaphore: fineRiempimento (simplified)


```
public void fineRiempimento() {
   mutex.p();
   zampilliLiberi++;   // uno zampillo liberato

   // valuta condizione per il rilascio di un cliente B (priorità)
   if (attesaB.queue()>0 && (attesaA.queue()==0 || stat!=0))
      attesaB.v();       // cede al cliente risvegliato mutex

   // valuta condizione per il rilascio di un cliente A
   else if (attesaA.queue()>0)
      attesaA.v();    // cede al cliente risvegliato mutex

   else
      // solo in questo caso rilascia la mutua esclusione
      mutex.v();
} //[m] fineRiempimento
```

Priority to B: its condition

Mutex transfer

Condition for A

Mutex transfer

No resume, release mutex

Solution with Critical Region - 1

- Often **one Region** is enough for each application class instance which can extend **Region** (though not necessarily)

- The condition of the enterWhen clause is the one permitting **the thread to enter the critical section**

Solution with Critical Region - 2

- We assume that the implementation guarantees the resuming of all the waiting processes in the same order as they arrived (based on semaphores with **FIFO** waiting queues)
- Shared variables must be updated in mutual exclusion:

  sometimes this requires a protected reservation section before entering the conditional region (using a not conditional **enterWhen**)

Critical Region: variables


```
public class FontanaReg {
    private static final long Riempimento = 15500L;
        // tempo di riempimento
    private int stat = 2;
        // conteggio per priorita` clienti B
    private int zampilliLiberi = 8;
        // i clienti in fontana saranno 8-zampilliLiberi
    private int ultimoZampillo = 7;
        // ultimo zampillo occupato (0..7)
    private int clientiA = 0, clientiB = 0;
        // clienti nelle rispettive code
    private Region ass = new Region(0);
        // protezione della sezione critica
```

Critical Region: entraCodaA - 1


```
public int entraCodaA() {

    ass.enterWhen();  // prenotazione
    clientiA++;
    System.out.println(
        "vvv Il cliente "+Thread.currentThread().getName()+
        " di tipo A attende in coda (clientiA="+clientiA+")");
    ass.leave();

    ass.enterWhen(new RegionCondition() {
        public boolean evaluate() {
            // verifica la condizione di risveglio di A
            return ! (zampilliLiberi==0 || (clientiB>0 && stat!=0) );
            }
    });
... ... ...
```

Condition for A

Critical Region: entraCodaA - 2


```
. . . . .
clientiA--;
System.out.println("^^^ Il cliente "+
      Thread.currentThread().getName()+
      " di tipo A termina l'attesa in coda (clientiA="+clientiA+")");
// assegna zampillo
zampilliLiberi--;
// reset di stat
stat=2;
System.out.println("************ zampilli liberi = "+
      zampilliLiberi);
int zamp = (ultimoZampillo = (ultimoZampillo+1)%8)+1;
ass.leave();
return zamp;
} //[m] entraCodaA
```

Critical Region: entraCodaB - 1


```
public int entraCodaB()  {
    ass.enterWhen();
    // prenotazione
    clientiB++;
    System.out.println("vvv Il cliente "+
        Thread.currentThread().getName()+
        " di tipo B attende in coda (clientiB="+clientiB+")");
    ass.leave();
    ass.enterWhen(new RegionCondition() {
        public boolean evaluate() {
            // verifica la condizione di risveglio di B
            return ! (zampilliLiberi==0 || (clientiA>0 && stat==0) );
            }
    });
```

Condition for B

Critical Region: entraCodaB - 2


```
clientiB--;
System.out.println("^^^ Il cliente "+
      Thread.currentThread().getName()+
      " di tipo B termina l'attesa in coda (clientiB="+clientiB+")");
// assegna zampillo
zampilliLiberi--;
System.out.println("************ zampilli liberi = "+
      zampilliLiberi);
if (clientiA>0)
      stat--;  // conteggio specifico
int zamp = (ultimoZampillo = (ultimoZampillo+1)%8)+1;
ass.leave();
return zamp;
} //[m] entraCodaB
```

Critical Region: fineRiempimento


```
public void fineRiempimento() {
    ass.enterWhen();
    zampilliLiberi++;
    // uno zampillo liberato
    System.out.println("************ zampilli liberi = "+
        zampilliLiberi);
    ass.leave();
    } //[m] fineRiempimento
```

Solution with Hoare's Monitor

- Often it is the more flexible solution
- On each **condition** variable, in a FIFO queue, some threads are waiting for a **specific condition**
- The semantics of a HM guarantees the transfer of mutex to the resumed thread
- Because the implementation is based on **semaphores with FIFO queue**, the requirements for a HM are fully assured

HM: variables


```
public class FontanaMon extends Monitor {
    private static final long Riempimento = 15500L;
        // tempo di riempimento
    private int stat = 2;
        // conteggio per priorità clienti B
    private int zampilliLiberi = 8;
        // i clienti in fontana saranno 8-zampilliLiberi
    private int ultimoZampillo = 7;
        // ultimo zampillo occupato (0..7)
    private int clientiA = 0, clientiB = 0;
        // clienti nelle rispettive code
    private Condition attesaA = new Condition();
    private Condition attesaB = new Condition();
        // clienti in attesa
```

HM: entraCodaA


```
public int entraCodaA() {
    mEnter();
    // verifica la condizione di attesa per A
    if (clientiA>0 || zampilliLiberi==0 || (clientiB>0 && stat!=0)) {
        clientiA++;           // deve attendere
        System.out.println("vvv Il cliente "+
            Thread.currentThread().getName()+
            " di tipo A attende in coda (clientiA="+clientiA+")");
        attesaA.cWait();
        clientiA--;
        System.out.println("^^^ Il cliente "+
            Thread.currentThread().getName()+
            " di tipo A termina l'attesa in coda (clientiA="+clientiA+")");
    }
    zampilliLiberi--;                // assegna zampillo in mutua esclusione
    stat=2;                          // reset di stat
    System.out.println("************ zampilli liberi = "+zampilliLiberi);
    int zamp = (ultimoZampillo = (ultimoZampillo+1)%8)+1;
    mExit();
    return zamp;
    } //[m] entraCodaA
```

HM: entraCodaB


```
public int entraCodaB() {
    mEnter();
    // verifica la condizione di attesa sul semaforo privato tipo B
    if (clientiB>0 || zampilliLiberi==0 || (clientiA>0 && stat==0) )   {
        clientiB++;            // deve attendere
        System.out.println("vvv Il cliente "+
            Thread.currentThread().getName()+
            " di tipo B attende in coda (clientiB="+clientiB+")");
        attesaB.cWait();
        clientiB--;
        System.out.println("^^^ Il cliente "+
            Thread.currentThread().getName()+
            " di tipo B termina l'attesa in coda (clientiB="+clientiB+")");   }
    zampilliLiberi--;          // assegna zampillo in mutua esclusione
    System.out.println("************ zampilli liberi = "+zampilliLiberi);
    if (clientiA>0)
        stat--;  // conteggio specifico
    int zamp = (ultimoZampillo = (ultimoZampillo+1)%8)+1;
    mExit();
    return zamp;
    } //[m] entraCodaB
```

HM: fineRiempimento


```
public void fineRiempimento()  {
    mEnter();
    zampilliLiberi++;
    // uno zampillo liberato
    System.out.println("************ zampilli liberi = "+zampilliLiberi);
    // valuta condizione per il rilascio di un cliente B
    // che ha priorita`
    if (clientiB>0 && (clientiA==0 || stat!=0) )
        // cede al cliente risvegliato la mutua esclusione
        attesaB.cSignal();
        // valuta condizione per il rilascio di un cliente A
    else if (clientiA>0)
        // cede al cliente risvegliato la mutua esclusione
        attesaA.cSignal();
    mExit();
    } //[m] fineRiempimento
```

Solution with Java Monitor

- It is more similar to a Critical Region, but the waiting condition must be inserted in a while clause and its body contains the wait() call
- In the release method we insert a **notifyAll()** call to make all the waiting threads repeat the evaluation of their (different) conditions
- For all the resumed threads must compete for the object lock with all the others, there is no assurance that the waiting order is maintained: if you need it, you can add a simple 'ticketing' system

JM: variables


```
public class FontanaJav {
    private static final long RIEMPIMENTO = 15500L;
        // tempo di riempimento
    private int stat = 2;
        // conteggio per priorità clienti B
    private int zampilliLiberi = 8;
        // i clienti in fontana saranno 8-zampilliLiberi
    private int ultimoZampillo = 7;
        // ultimo zampillo occupato (0..7)
    private int clientiA = 0, clientiB = 0;
        // clienti nelle rispettive code
    private int ticketA=0, ticketB=0;
    private int servizioA=0, servizioB=0;
        // per assicurare l'ordine
```

JM: entraCodaA


```
public synchronized int entraCodaA() {
    clientiA++;
    System.out.println("vvv Il cliente "+Thread.currentThread().getName()+
        " di tipo A attende in coda (clientiA="+clientiA+")");
    int ticket = ticketA++;
    // ripete l'attesa su condizione
    while(zampilliLiberi==0 || (clientiB>0 && stat!=0) || servizioA != ticket)
        try { wait(); } catch (InterruptedException e) {};
    clientiA--;
    System.out.println("^^^ Il cliente "+Thread.currentThread().getName()+
        " di tipo A termina l'attesa in coda (clientiA="+clientiA+")");
    zampilliLiberi--;          // assegna zampillo
    stat=2;                    // reset di stat
    System.out.println("************ zampilli liberi = "+zampilliLiberi);
    servizioA++;
    return (ultimoZampillo = (ultimoZampillo+1)%8)+1;
} //[m] entraCodaA
```

JM: entraCodaB

```
public synchronized int entraCodaB()  {
    clientiB++;
    System.out.println("vvv Il cliente "+Thread.currentThread().getName()+
        " di tipo B attende in coda (clieniB="+clientiB+")");
    int ticket = ticketB++;
    // ripete l'attesa su condizione
    while(zampilliLiberi==0 || (clientiA>0 && stat==0) || servizioB != ticket)
        try { wait(); } catch (InterruptedException e) {};
    clientiB--;
    System.out.println("^^^ Il cliente "+Thread.currentThread().getName()+
        " di tipo B termina l'attesa in coda (clientiB="+clientiB+")");
    zampilliLiberi--;           // assegna zampillo
    System.out.println("************ zampilli liberi = "+zampilliLiberi);
    if (clientiA>0)
        stat--;                 // conteggio specifico
    servizioB++;
    return (ultimoZampillo = (ultimoZampillo+1)%8)+1;
    } //[m] entraCodaB
```

JM: fineRiempimento


```
public synchronized void fineRiempimento() {
    zampilliLiberi++;
    // uno zampillo liberato
    System.out.println("************ zampilli liberi = "+
        zampilliLiberi);
    notifyAll();
    } //[m] fineRiempimento
```

Solution with ADA-Java Monitor

- Now the synchronization is provided by a server task with selective waits, two guarded **codaA(out: int zamp)** and **codaB(out: int zamp)** representing the two waiting queues and returning the index of the assigned spout, and **uscita()** called when the customer leaves the fountain
- All the state variables are accessed only by the server (which serializes all necessary accesses)
- The filling is simulated by a sleep within the client thread
- Relative priority is given by controlling the opening and closing of guards

AJ: variables


```
public class FontanaADAAll extends ADAThread implements
FontanaADAAllStr {
     private static final long SERVTMO = 2000L;
      // timeout vari
    private static final long RIEMPIMENTO = 15500L;
      // tempo di riempimento
    private int clientiA = 0, clientiB = 0;
      // clienti nelle rispettive code
    private int stat = 2;
      // conteggio per priorita` clienti B
    private int zampilliLiberi = 8;
      // i clienti in fontana saranno 8-zampilliLiberi
    private int ultimoZampillo = 7;
      // ultimo zampillo occupato (0..7)
```

AJ: codaA

esempio d'uso della macro semplificativa:

```
      sel.add( when (zampilliLiberi>0 && (entryCount(codaBStr)==0
        || stat==0) =>
        codaA[out: int zamp]
        {
          // parametro di input non significativo
          zampilliLiberi--;
          // reset di stat
          stat=2;
          System.out.println("************ zampilli liberi =
"+zampilliLiberi);
          // assegna zampillo
          zamp = (ultimoZampillo = (ultimoZampillo+1)%8)+1;
        }
        );
```

AJ: codaB


```
// entry codaB, choice=1
      sel.add(new Guard() {
          public boolean when() {
              return zampilliLiberi!=0 &&
                (entryCount(codaAStr)==0 || stat!=0);
          } //[m] when
        } /*{c} <anonim>*/, codaBStr,
        new Entry() {
          public Object exec(Object inp) {
              zampilliLiberi--;
              System.out.println("************* zampilli liberi =
"+zampilliLiberi);
              if (entryCount(codaAStr)>0)
                  stat--;  // conteggio specifico
              // assegna zampillo
              int zamp = (ultimoZampillo = (ultimoZampillo+1)%8)+1;
              return new Integer(zamp);
          }
        } /*{c} <anonim> */ );
```

AJ: uscita


```
// entry uscita, choice=2, nessuna guardia
       sel.add(uscitaStr,
         new Entry()
         {
           public Object exec(Object inp)
           {
               zampilliLiberi++;
                 // uno zampillo liberato
               System.out.println("************* zampilli liberi = "+zampilliLiberi);
               return null;
           }
         } //{c} <anonim>
         );
```

AJ: extra


```
// entry stato, choice=3
      sel.add(new Guard()
        {
          public boolean when()
          {
              return zampilliLiberi<=1;
          } //[m] when
        } //{c} <anonim>
        , SERVTMO  // delay costante
      );

      while (true)
      {
          int choice = sel.accept();
          System.out.println("[[[entry "+sel.choice2Str(choice));
          switch(choice)
          {
            case 3:
              System.out.println("[[!!!! SERVER TIMEOUT zampilli liberi="+zampilliLiberi);
              break;
            default:
              // nulla
          }
      }
```

Variants

- With the current times, an accumulation in A is probable:increase the average arrival time (for example enlarge the range between 0.5 s and 3.5 s)
- Set the mug filling time variable within a given range
- Threads are coded in only one class external to the application class and the type (A or B) is given as a parameter in the constructor
- The solution with Hoare's Monitor **does not extend Monitor** but uses an instance within the application class

The end

aa.1

The Fontebella baths

Semaphores, Regions, Monitors