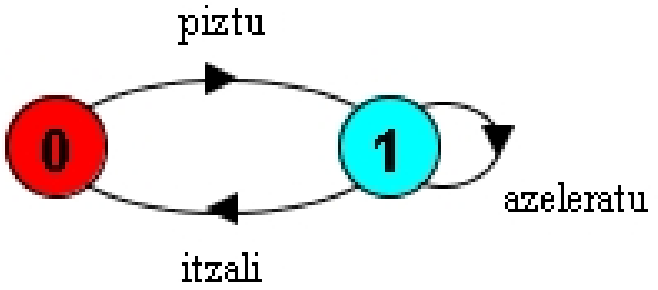
Prozesuak eta hariak

(Egoera-ereduak eta Java programak)

Egoera-ereduak eta Java programak

Eredua: munduaren adierazpide sinplifikatua

- Ereduak erabiliko ditugu proposatutako diseinuak ea baliozkoak eta egokiak diren aztertzeko.
- Ereduaren fokua: **konkurrentzia**.
- Ereduak deskribatuko ditugu egoera finituetako makinekin.



Java lengoaia erabiliko dugu programa konkurrenteak inplementatzeko

- Hariak (Threads)

Prozesu konkurrenteak

Prozesua:
ekintza atomikoen sekuentzia.

Prozesuak modelatuko ditugu
egoera finituetako makinekin.

Prozesuak Java-z programatuko ditugu
hariak (threads) erabiliz.

Prozesuak eta hariak

Ereduak:

Egoera finituetako prozesuak (FSP)

prozesuak modelatzeko ekintza atomikoen sekuentziak bezala.

Etiketaturiko trantsizio-sistemak (LTS)

analizatzeko, bisualizatzeko eta portaeraren animazioa lortzeko

Inplementazioa:

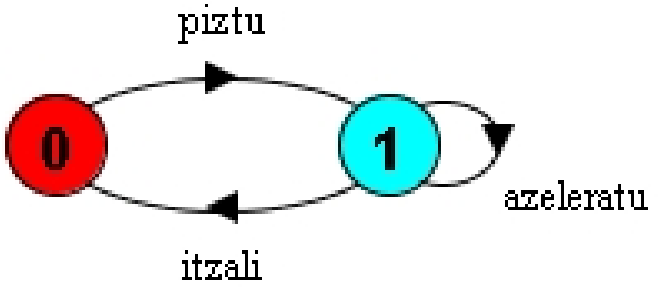
Java-ko hariak (threads)

2.1 Prozesuak modelatzen

- Ereduak egoera-makinekin deskribatuko ditugu:

  Etiketaturiko Trantsizio Sistemak

  (Labelled Transition Systems, **LTS**).



- Modu testualean deskribatzeko:

  Egoera Finituetako Prozesuak

  (Finite State Processes, **FSP**)

- Bisualizatzeko, analizatzeko eta egiaztapen mekanikoa egiteko: ***LTSA*** analisi-tresna.

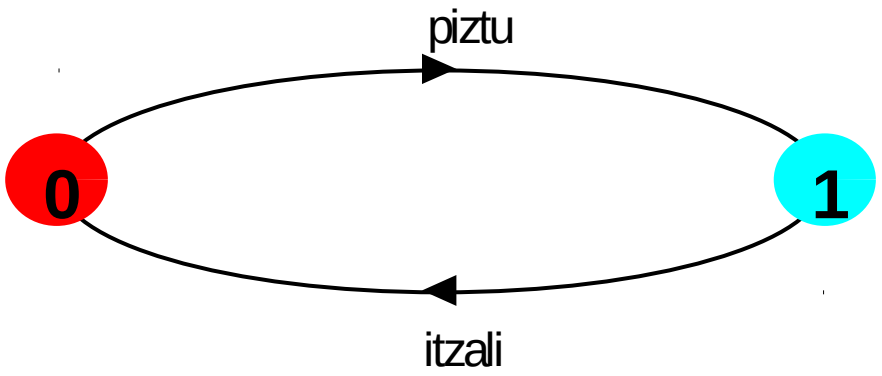
- LTS – forma grafikoa
- FSP - forma algebraikoa

Prozesuak modelatzen

Prozesu bat programa sekuentzial baten exekuzioa da.

Eredua ➔ Egoera finituetako makina

(ekintza atomikoen sekuentzia bat egikaritzen egoeratik egoerara igarotzen da)



Argi etengailua

LTS

`piztu→itzali→piztu→itzali→...`

***traza**:* ekintzen sekuentzia

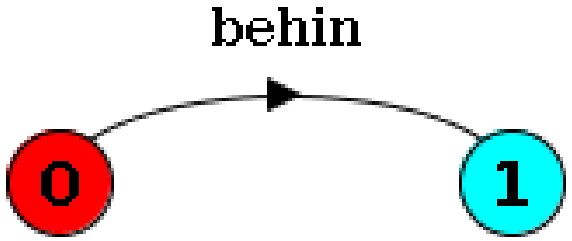
Ekintza atomiko bakoitzak

uneko egoeratik hurrengo egoerara trantsizio bat eragiten du.

FSP – ekintza-aurrizkia (action prefix)

Bitez **x** ekintza bat eta **P** prozesu bat, orduan
(x->P) prozesuak
x ekintzarekin hasi, eta ondoren
P-k deskribatzen duena egiten du

BEHINEGIN = (behin -> STOP).



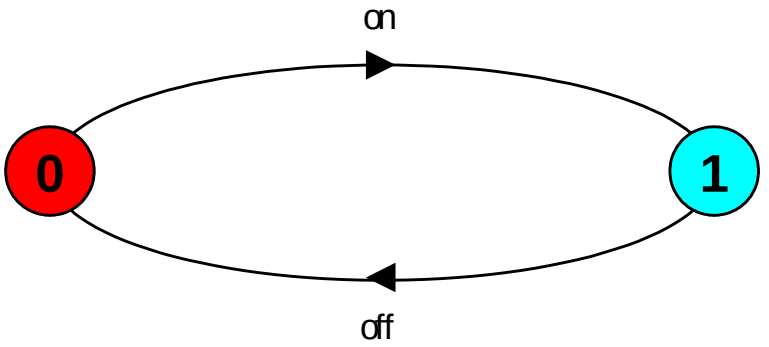
BEHINEGIN egoera-makina
(prozesu terminala)

- **ekintzak** minuskulaz
- **PROZESUAK** maiuskulaz

FSP – ekintza-aurrizkia eta errekurtsioa

Portaera errepikakorrak errekurtsioa erabiltzen du:

```
ETENGAILUA = OFF,
OFF = (on -> ON),
ON  = (off-> OFF).
```



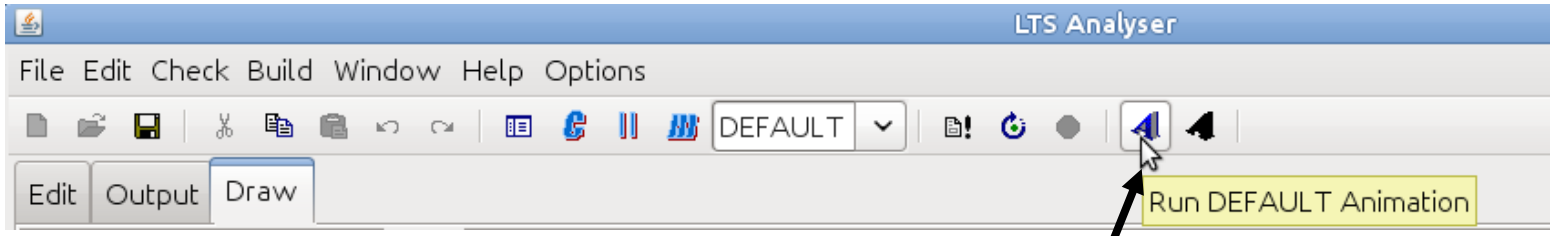
Definizio trinkoagoa lortzearren ordezkatzen dugu:

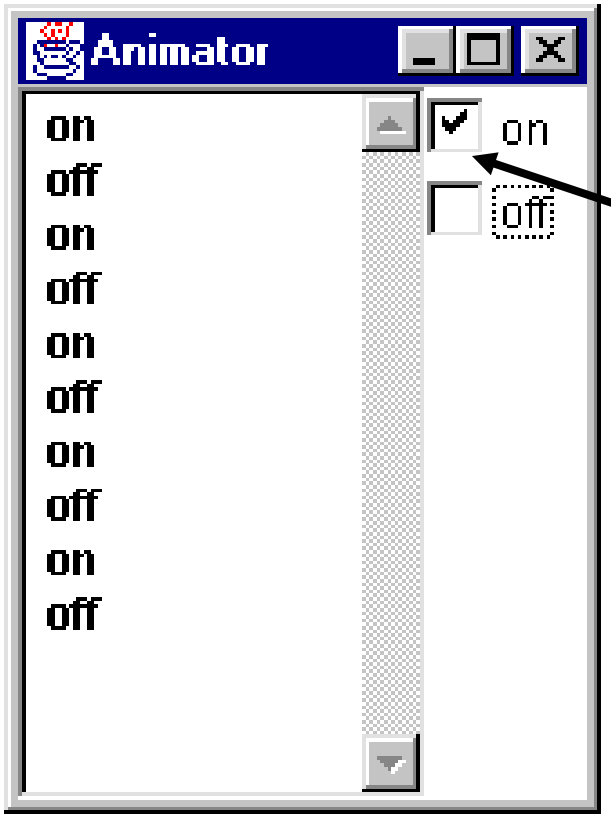
```
ETENGAILUA = OFF,
OFF = (on ->(off->OFF)).
```

Eta berriz:

```
ETENGAILUA = (on->off-> ETENGAILUA).
```

Animazioa LTSA erabiliz

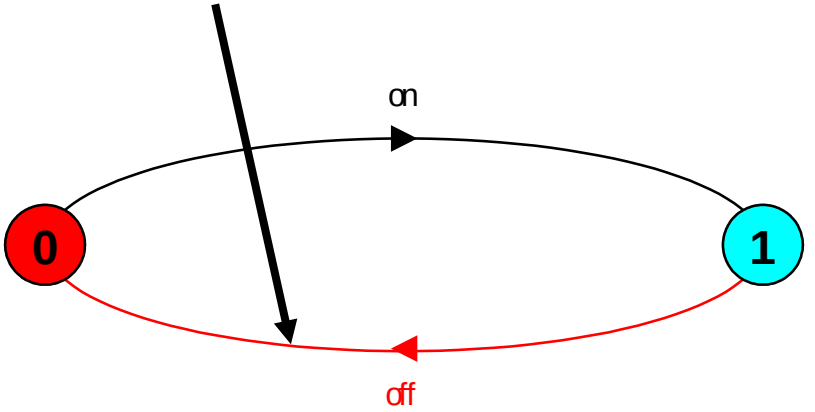




Traza erakusteko *LTSA animator* erabili dezakegu.

Markatutako ekintzak hautagarriak dira aukera egitean.

Azkeneko ekintza **gorriz** adierazten da.

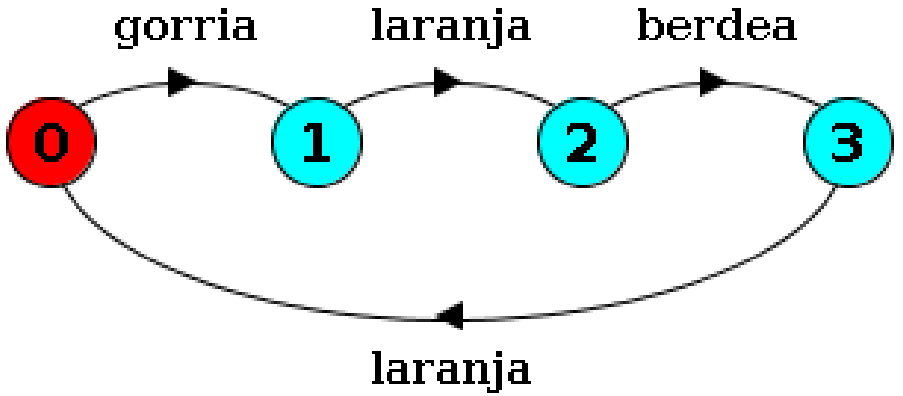


FSP – ekintza aurrizkia

Semaforo baten FSP eredua:

```
SEMAFOROA = (gorria->laranja->berdea->laranja
                -> SEMAFOROA).
```

LTSA-k sortzen duen LTS-a:



Traza:

gorria→laranja→berdea→laranja→gorria→laranja→berdea…

FSP - aukera

Bitez **x** eta **y** ekintzak, eta **P** eta **Q** prozesuak, orduan
(x-> P | y-> Q) prozesuak
x edo **y** ekintzekin hasi
eta ondoren
P egingo du, lehenengo ekintza **x** bazen, eta
Q egingo du, lehenengo ekintza **y** bazen.

Nork edo zerk egiten du aukera?

Izan daiteke ingurunea edo baita barne prozesu bat ere.

Ba al dago ezberdintasunik sarrera eta irteerazko ekintzen artean?

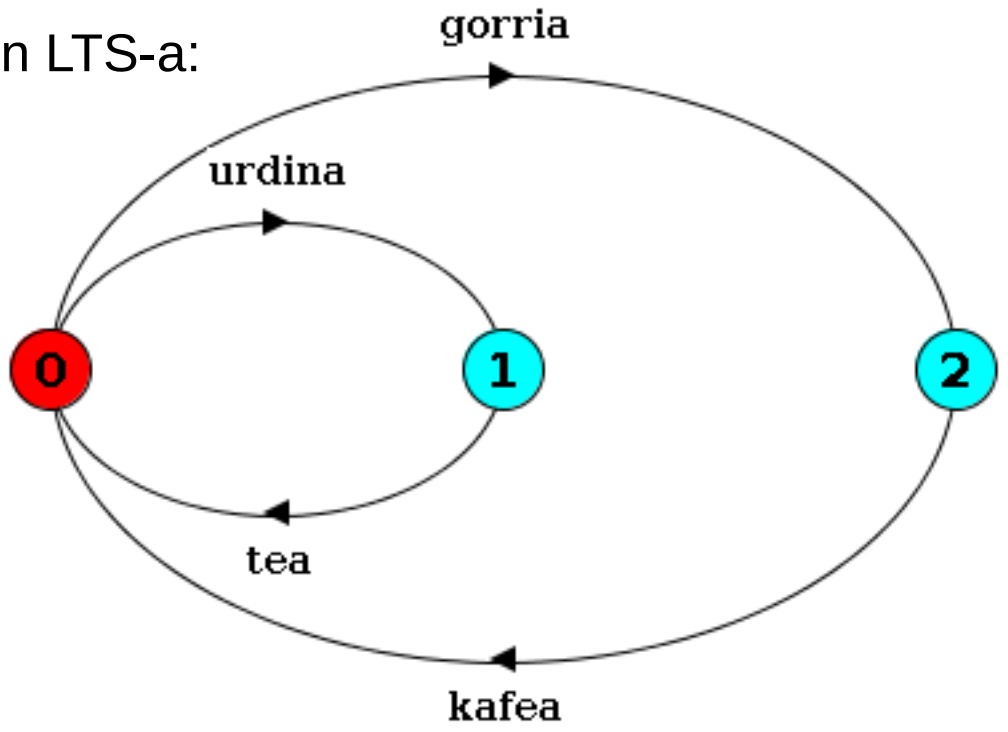
Gure ereduetan ez da ezberdintasun semantikorik izango.
Hala ere, normalean sarrera-ekintzek aukeretan parte hartuko dute eta irteera-ekintzek ez.

FSP - aukera

Edari-makina baten FSP eredua:

```
EDARIAK = (gorria ->kafea->EDARIAK
          |urdina ->tea  ->EDARIAK
          ).
```

LTSA -k sortzen duen LTS-a:



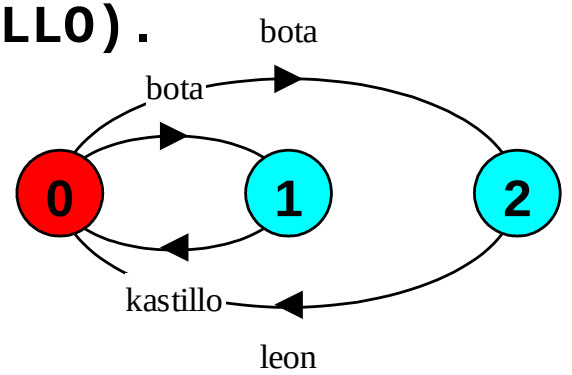
Arik: Traza posibleak?

Aukera ez-determinista

(x-> P | x -> Q) prozesua
x ekintzarekin hasten da,
eta gero
P edo **Q** egingo du.

Txanpon bat botatzen:

```
LEONKASTILLO   = (bota->LEON|bota->KASTILLO),
LEON           = (leon->LEONKASTILLO),
KASTILLO       = (kastillo->LEONKASTILLO).
```



Beste modu batez:

```
LEONKASTILLO   = (bota->leon->LEONKASTILLO
                 |bota->kastillo->LEONKASTILLO).
```

Edo:

```
LEONKASTILLO   = (bota->{leon,kastillo}->LEONKASTILLO).
```

Arik: Traza posibleak?

Huts-egitea modelatzen

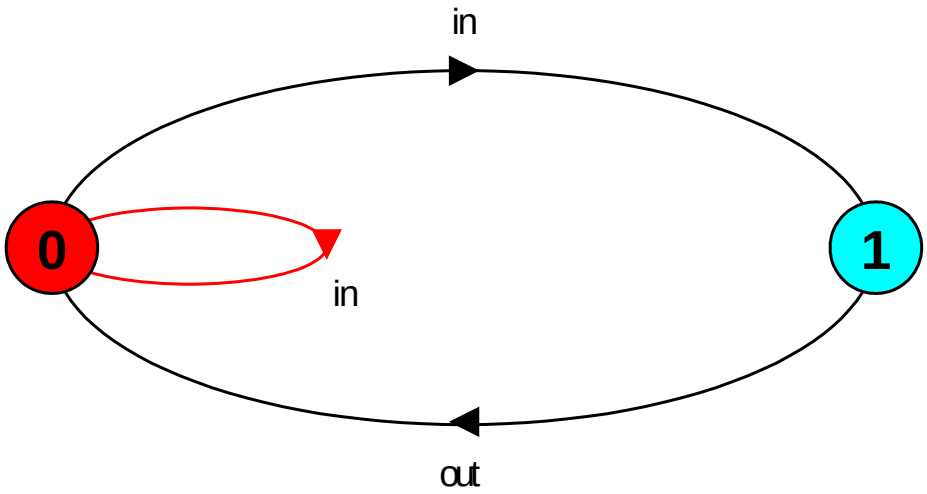
Modelatu nahi dugu honelako komunikazio kanal ez fidagarri bat:

- **in** ekintzak onartzen ditu,
- huts egiten badu ez du irteerarik ematen,
- bestela **out** ekintza burutzen du.

Nola?

Ez-determinismoa erabiliz…

```
KANALA = (in->KANALA
         |in->out->KANALA
         ).
```



Ekinten-multzoak

Ekintza desberdinek jokarea berdina baldin badute,
ekintza multzo bat erabili ahal dugu, aukera esplizitua erabili beharrean.

Adibidez, ondoko prozesuak baliokideak dira eta LTS berdina sortzen dute:

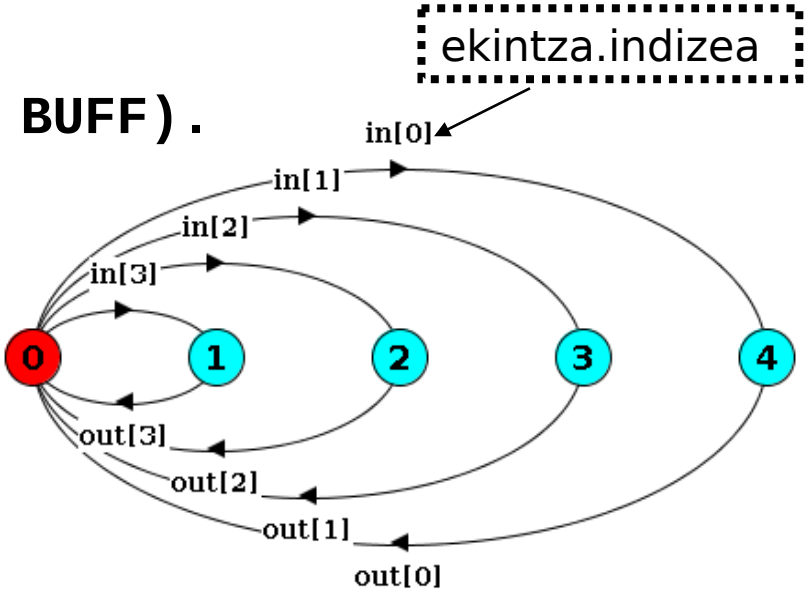
ATEA1 = (ireki -> ATEA1
| itxi -> ATEA1).

ATEA2 = ({ireki, itxi} -> ATEA2).

FSP – prozesu indexatuak eta ekintzak

Buffer batek jasotzen du 0 eta 3 bitarteko balio bat eta ondoren balio bera itzultzen du:

```
BUFF = (in[i:0..3]->out[i]-> BUFF).
```



Honen baliokidea da:

```
BUFF = (in[0]->out[0]->BUFF
       |in[1]->out[1]->BUFF
       |in[2]->out[2]->BUFF
       |in[3]->out[3]->BUFF
       ).
```

Edo balio lehenetsia duen prozesu-parametro bat erabiliz:

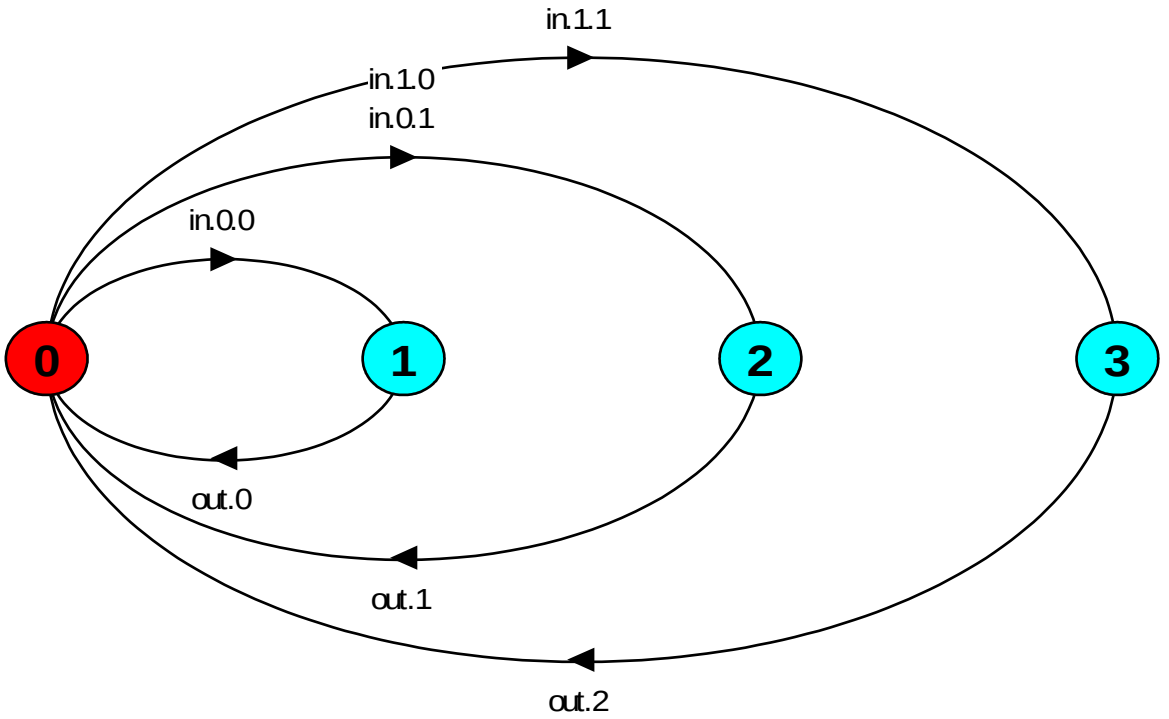
```
BUFF(N=3) = (in[i:0..N]->out[i]-> BUFF).
```

Edo, hobeto, konstante bat eta hein bat definituz eta erabiliz:

```
const N=3
range R=0..N
BUFF = (in[i:R]->out[i]-> BUFF).
```

FSP – konstante eta balio-eremuaren erazagupena

Bi balio sartu
eta batura itzuli:




```
const N = 1
range T = 0..N
range R = 0..2*N
```

```
BATURA          = (in[a:T][b:T]->GUZTIRA[a+b]),
GUZTIRA[s:R]    = (out[s]->BATURA).
```

edo

```
BATURA          = (in[a:T][b:T]-> out[a+b]-> BATURA).
```

FSP – baldintzak

(if B then (x -> P) else (y -> Q)) esan nahi du:

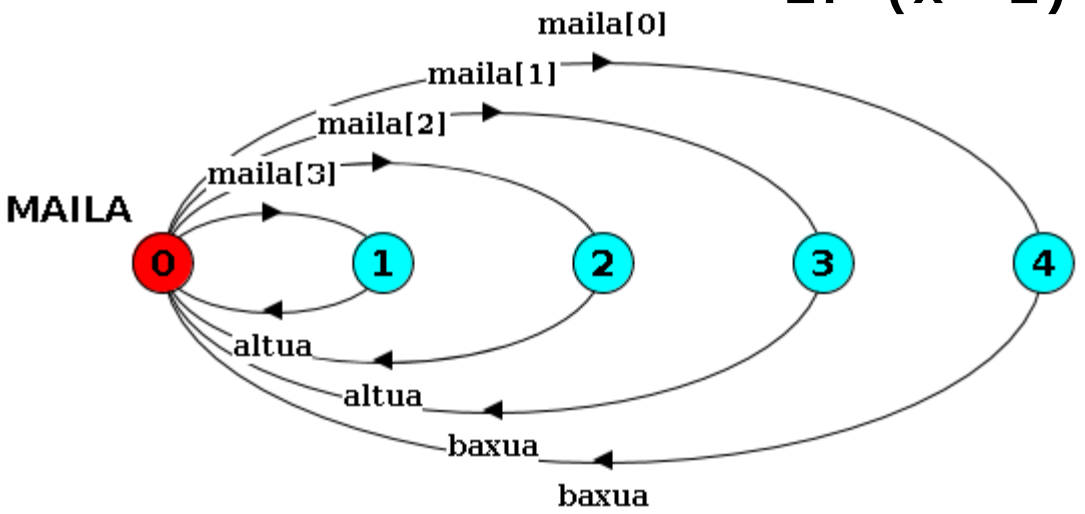
B baldintza egia bada,

x ekintza eta **P** egingo da

eta bestela, **B** faltsua bada,

y ekintza eta **Q** egingo da.

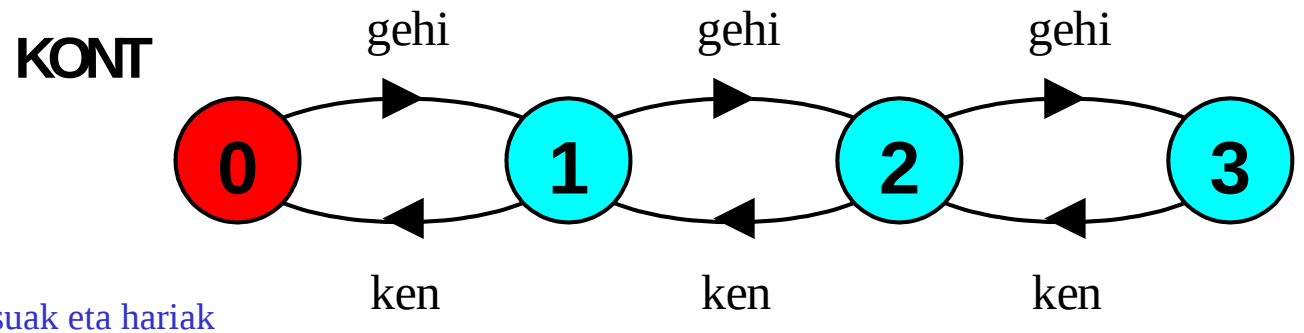
```
const N = 3
range R = 0..N
MAILA = (maila[x:0..3] ->
           if (x>=2) then (altua -> MAILA)
                     else (baxua -> MAILA) ).
```



FSP – ekintza babestuak (*guarded actions*)

(when B x -> P | y -> Q) aukerak esan nahi du:
B baldintza egia bada,
x zein **y** ekintzak hautagarriak dira,
bestela **B** faltsua bada,
x ekintza ezin da aukeratu.

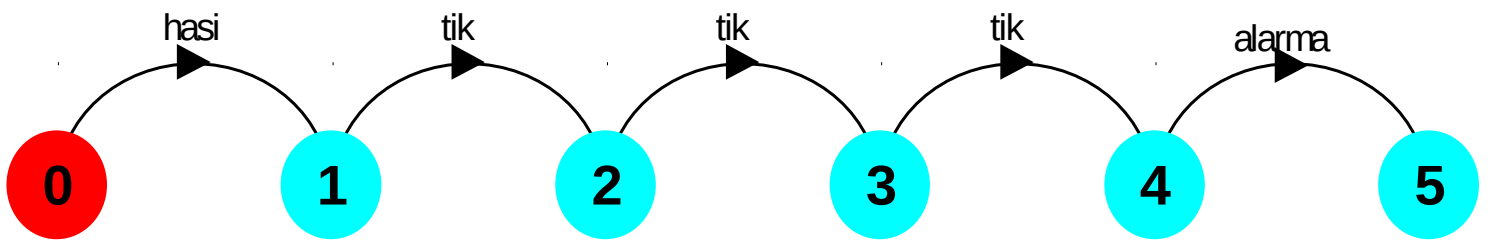
```
const N = 3
range R = 0..N
KONT      = KONT[0],
KONT[i:R] = ( when (i<N) gehi -> KONT[i+1]
            | when (i>0) ken  -> KONT[i-1]
            ).
```



FSP – ekintza babestuak

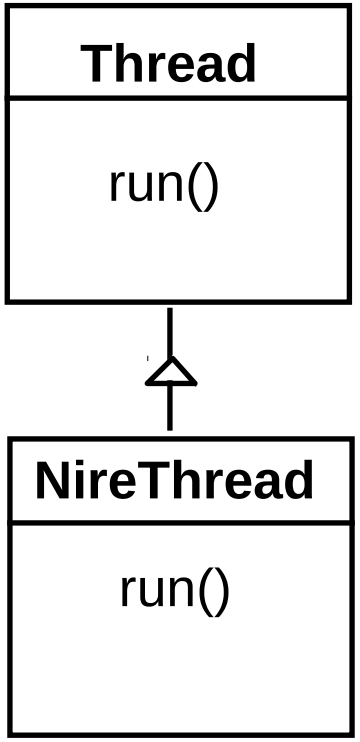
Atzekoz aurrera zenbatzen duen kronometro bat,
N tik eta gero alarma jotzen duena.

```
const N = 3
range R = 0..N
KRONO       = (hasi->KRONO[N]),
KRONO[i:R] = (  when(i>0) tik    -> KRONO[i-1]
              | when(i==0)alarma -> STOP
              ).
```



Java-ko hariak sortzeko bi modu: I

Thread klasearen azpiklase bat erazagutuz.



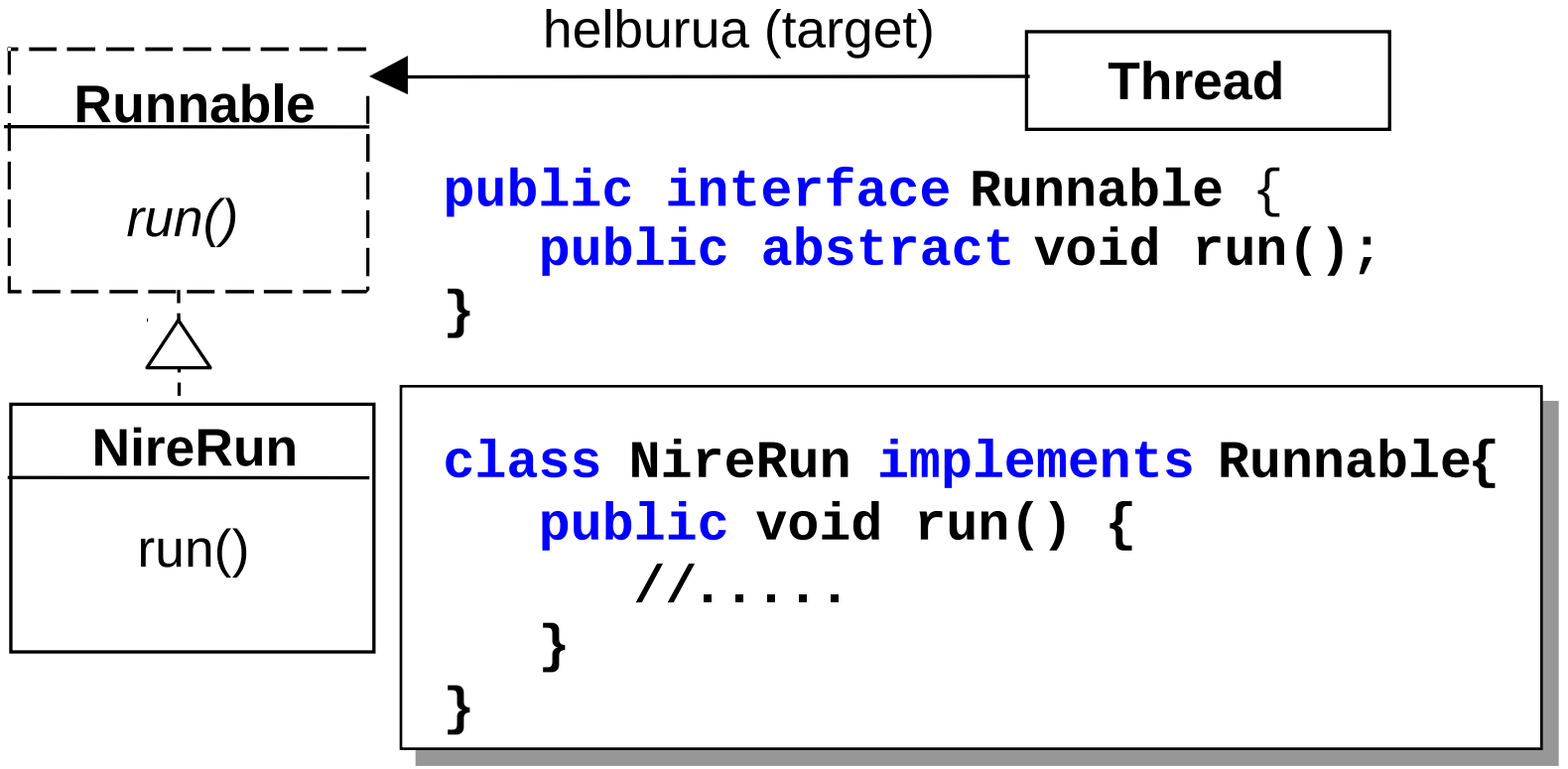
Egikaritu nahi dugun kodea, azpiklaseko run() metodoan inplementatuko dugu.

```
class NireThread extends Thread {
      public void run() {
            //......
      }
}
```

Java-ko hariak sortzeko bi modu: II

Runnable interfazea inplementatzen duen klase bat erazagutuz.

- Klasea beste klase baten azpiklasea denean, ezin dugu Thread-en azpiklasea egin, Java-k ez duelako herentzia anitza onartzen.

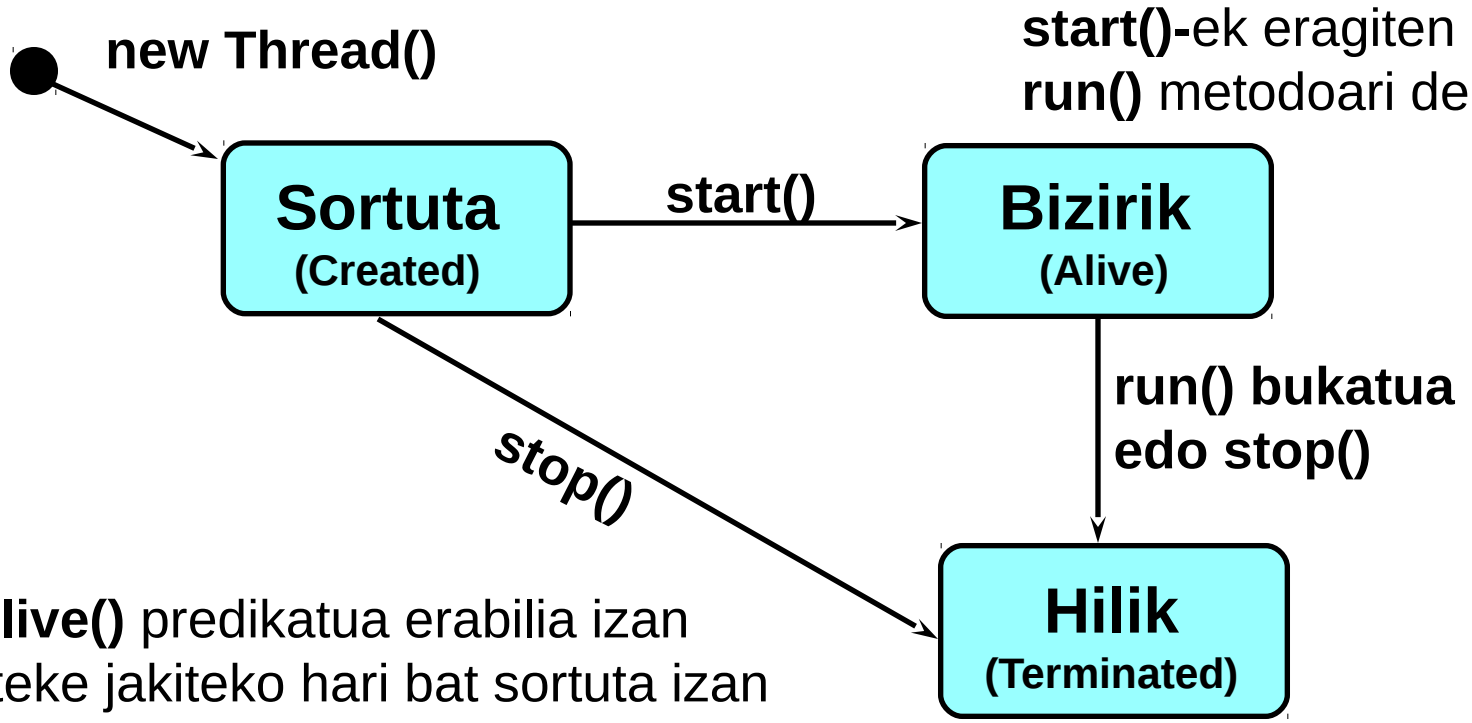



```
public interface Runnable {
    public abstract void run();
}
```

```
class NireRun implements Runnable{
    public void run() {
        //.....
    }
}
```

Java-ko harien bizi-zikloa

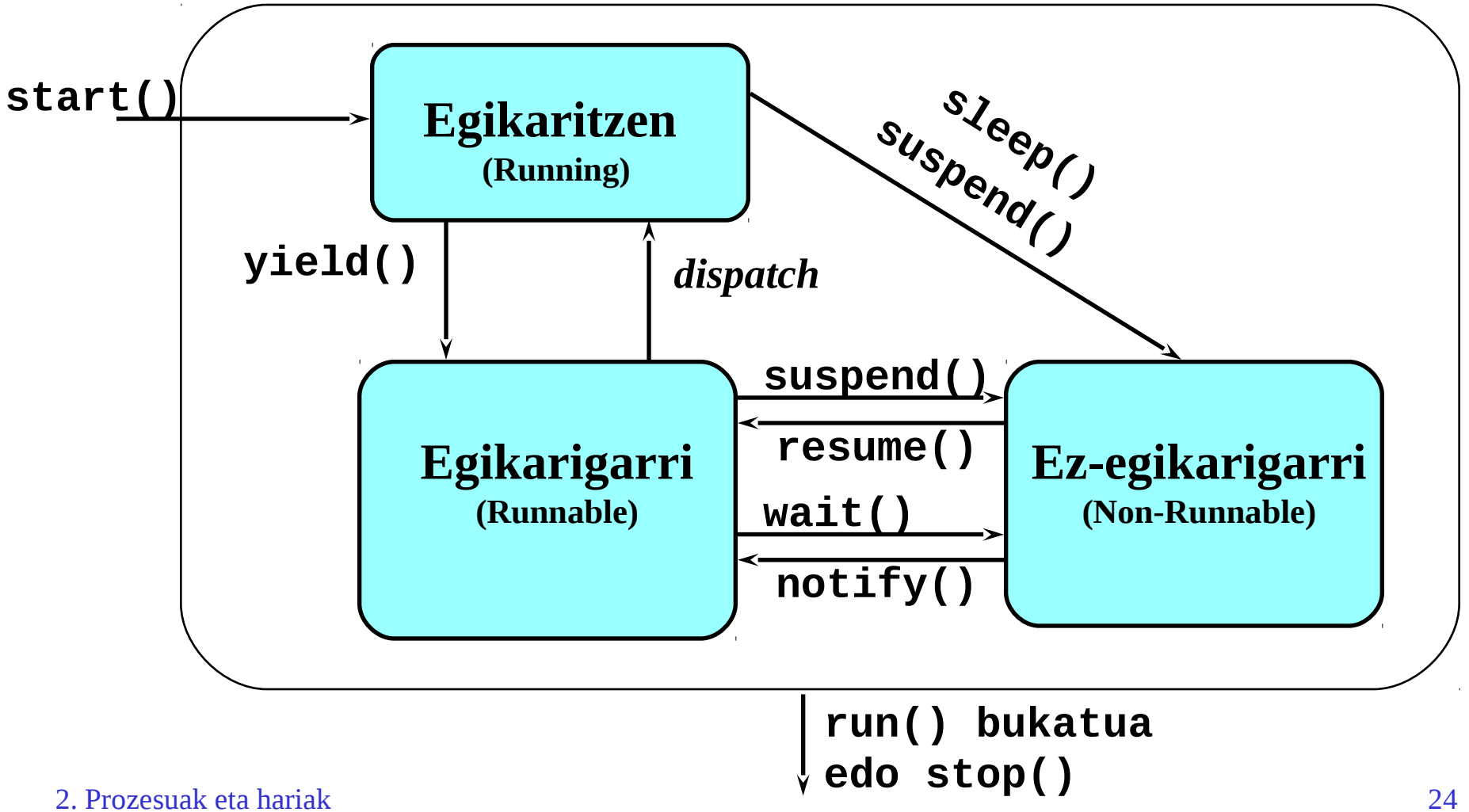
Hari baten bizi-zikloaren ikuspegia, egoera-trantsizioekin adierazita:



isAlive() predikatua erabilia izan daiteke jakiteko hari bat sortuta izan den baina ez hilik. Behin hilik, ezin da berriz start egin.

Java-ko hariak bizirik: egoera posibleak

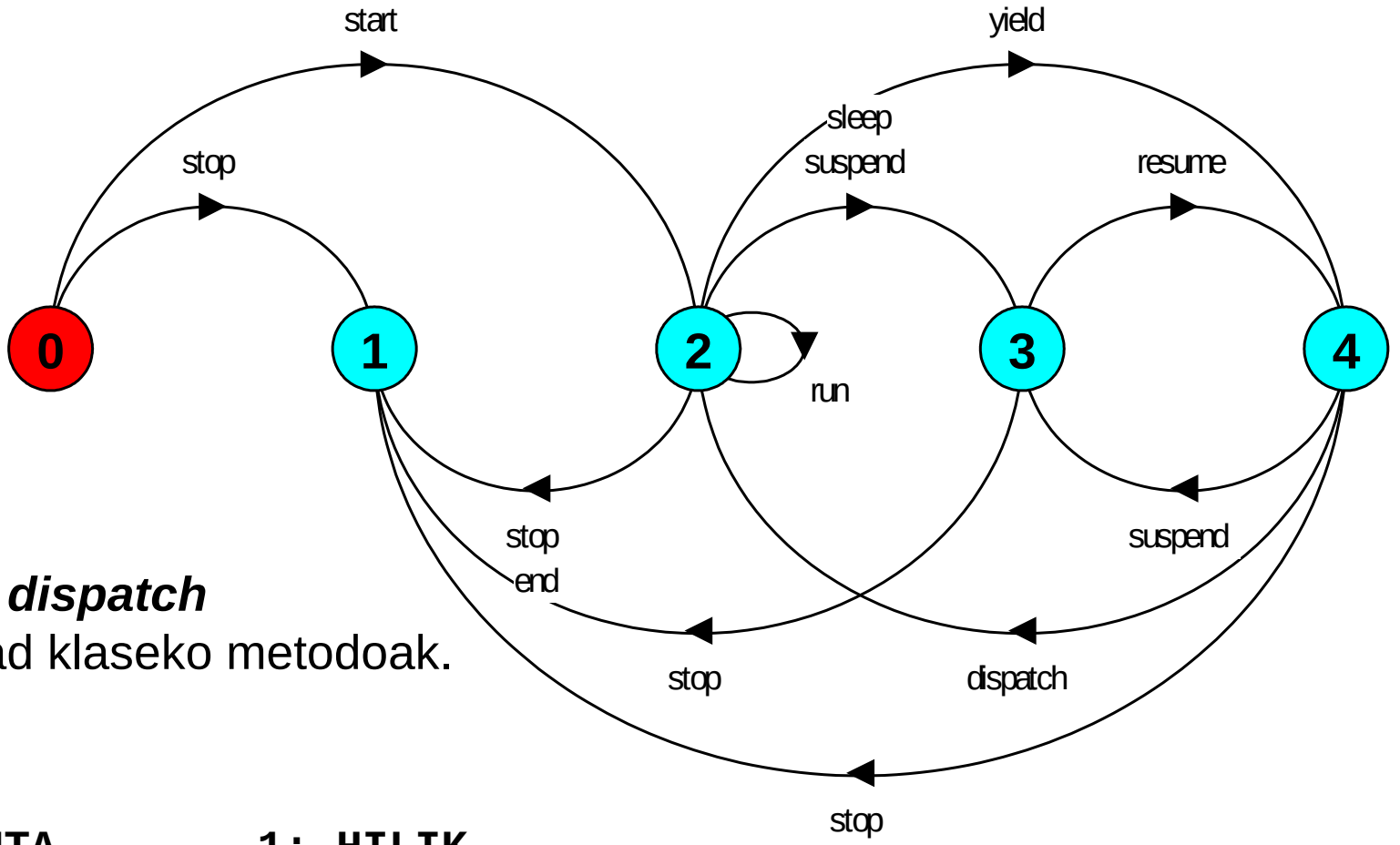
Behin **start()** egina, **bizirik** dagoen hari baten egoera posibleak:



Java-ko harien bizi-zikloa - FSP espezifikazioa

```
THREAD           = SORTUTA,
SORTUTA          = (start              ->EGIKARITZEN
                   |stop               ->HILIK),
EGIKARITZEN      = ({suspend,sleep}    ->EZ_EGIKARIGARRI
                   |yield              ->EGIKARIGARRI
                   |{stop,end}         ->HILIK
                   |run                ->EGIKARITZEN ),
EGIKARIGARRI     = (suspend            ->EZ_EGIKARIGARRI
                   |dispatch           ->EGIKARITZEN
                   |stop               ->HILIK ),
EZ_EGIKARIGARRI= (resume               ->EGIKARIGARRI
                   |stop               ->HILIK ),
HILIK = STOP.
```

Java-ko harien bizi-zikloa - FSP espezifikazioa



end, run eta ***dispatch***
ez dira Thread klaseko metodoak.

Egoerak:

```
0: SORTUTA          1: HILIK
2: EGIKARITZEN      3: EZ_EGIKARIGARRI     4 : EGIKARIGARRI
```

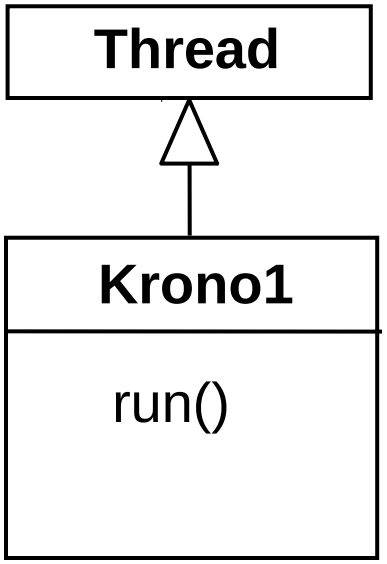
Kronometroaren adibidea

```
const N = 3
range R = 0..N
KRONO      = ( hasi->KRONO[N]),
KRONO[i:R] = ( when(i>0)  tik   -> KRONO[i-1]
             | when(i==0) alarma-> STOP
             ).
```

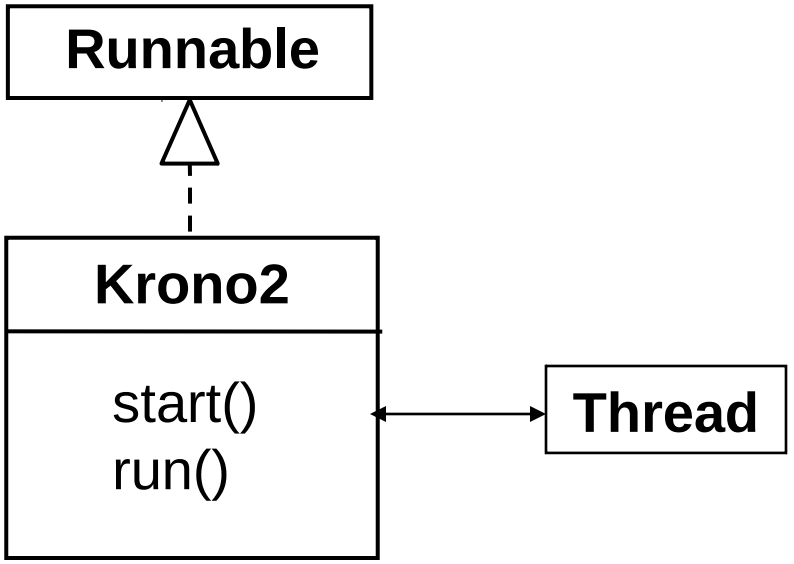
Bi modutan inplementatuko dugu:

- Thread-en azpiklase bat erazagutuz, eta
- Runnable inplementatzen duen klase bat erazagutuz.

Kronometroaren klase-diagramak



Krono1 klasea **Thread** klasearen azpiklasea da.



Krono2 klasea **Runnable** interfazea inplementatzen du.

Krono1 klasea

```
class Krono1 extends Thread {
  int i=10;
  public void run() {
     while (true) {
          if (i>0) {tik(); --i;}
          if (i==0){alarma(); return;}
     }
  }
  private void tik() {...}
  private void alarma() {...}
}

class KronoApp {
  public static void main (String args[]){
  Krono1 k = new Krono1();
  k.start();
  }
}
```

Thread klasearen azpiklasea da (**extends**).

run() metodoak egikaritu nahi dugun kodea izango du.

Haria hilko da **run()** bukatzean.

Erazagutu dugun klaseko objektu bat sortzen dugu **new()** eginez.

Hasieratzen dugu **start()** erabiliz

Krono1 klasea...

tik() eta **alarma()** metodoen inplementazioa:

```
private void tik() {
      System.out.println(i);
      try {sleep(1000);}
      catch (InterruptedException e) {}
}

private void alarma() {
      System.out.println("Bukatu da!");
}
```

Krono1 klasea...(b)

Honela ere inplementa genezake:

```
class Krono1b extends Thread {
   public void run()  {
      int i;
      for (i=10;i>=0;i--) {
            System.out.println(i);
            try {sleep(1000);}
            catch (InterruptedException e) {}
      }
      System.out.println("Bukatu da!");
   }
}
```

Krono2 klasea

```
class Krono2 implements Runnable{
  Thread t;

  public void start() {
     t = new Thread(this);
     t.start();
  }

  public void run() {
     int i;
     for (i=10;i>=0;i--) {
       System.out.println(i);
       try {Thread.sleep(1000);}
       catch (InterruptedException e){}
     }
     System.out.println("Bukatua!");
  }
}
```

Runnable interfazea implementatzen du.

Thread klaseko atributu bat sortu.

start() berrerazagutu.

Thread klaseko atributua instantziatu, bere eraikitzaileari deituz, klase-instantzia bera (this) argumentu gisa pasatuz.

Threada hasieratu

run() metodoa berrerazagutu.

sleep() metodoak erabiltzeko **Thread** klasekoa dela adierazi behar dugu.

Kronometro2 klasea

Krono2 Thread gisa instantziatua eta hasieratua izan daiteke:

```
class KronoApp {
  public static void main (String args[]){
  Krono2 k = new Krono2();
  k.start();
  }
}
```

Krono2 klasea **Applet**-en azpiklasea izan zitekeen:

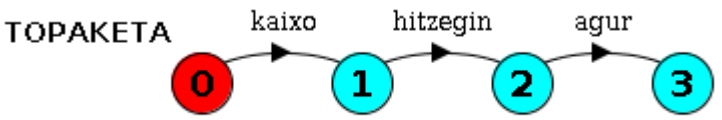
```
public class Krono2 extends Applet
                    implements Runnable {
  ...
}
```

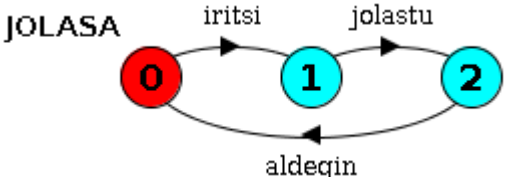
Krono2 klasearen azpiklase-hierarkia osoko klaseak ere **Thread** klasearen ezaugarriak edukiko dituzte:

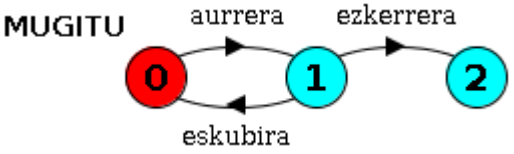
```
class Krono3 extends Krono2 {
}
```

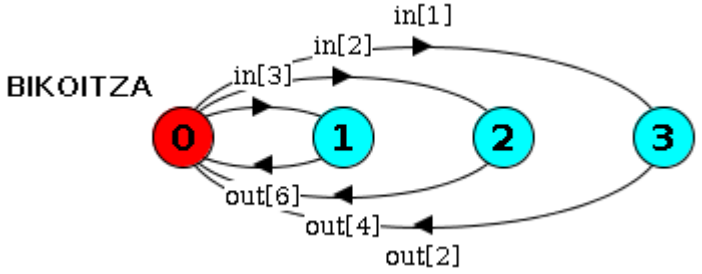
Ariketak (I)

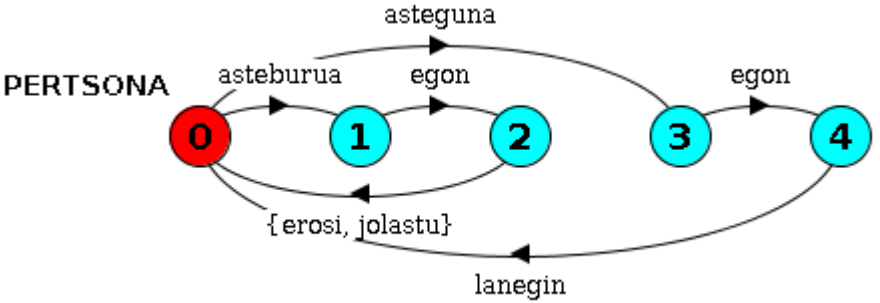
1. LTSak emanik FSPak lortu

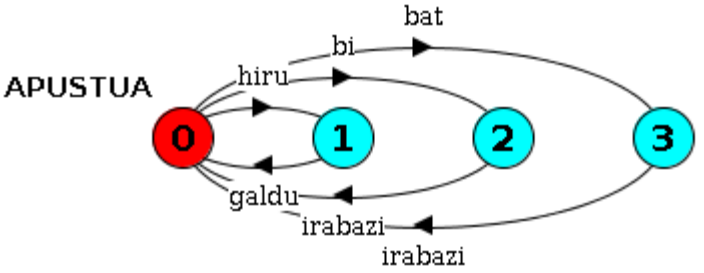


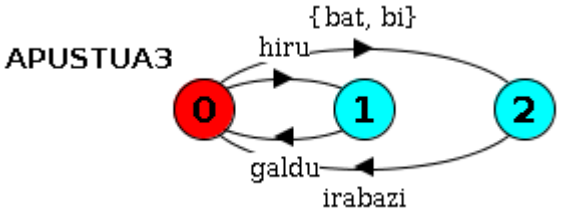












Ariketak (II)

Ondoko ariketen FSP eredua eman:

2. Aldagai batek 0..N heinean dauden balioak gordetzen ditu eta irakurri eta idatzi ekintzak onartzen ditu. N=2 dela suposa dezakegu.
   (Traza posible bat: idatzi[0]->irakurri[0]->irakurri[0]->idatzi[1]->idatzi[2]->irakurri[2]->...)

3. Makina batean botoi bat sakatzean 1 eta 0 balioak itzultzen dizkigu alternatiboki.
   (Traza: sakatu->1->sakatu->0->sakatu->1->sakatu->0…).

4. Bi zenbaki eman eta bietako maximoa itzuli.
   (Traza posible bat: in[1][3]->max[3]->in[2][0]->max[2]->in[1][1]->max[1]->...)

5. Igogailu bat. Gora eta behera egin dezake, 1. eta 6. pisuen artean.
   (Traza posible bat: non[4]->gora->non[5]-->gora->non[6]>behera->non[5]->...)

6. Urtegia. Uraren maila 0..9 artean egon behar du. Hasieran 5ean dago. 2 baino txikiagoa denean “gutxi” adierazten du, eta 8 baino handiagoa denean “asko”. Bestela “ongi”.
   (Traza posible bat: ...->ongi[7]->igo->ongi[8]->igo->asko[9]->jeitsi->ongi[8]->...)

7. Freskagarrien makina. Lata batek 15 zentimo balio du. Makinak onartzen ditu 5, 10 eta 20 zentimoko txanponak, eta bueltak ematen ditu.
   (Traza posible bat: [10]->[20]->lata->buetak[15])