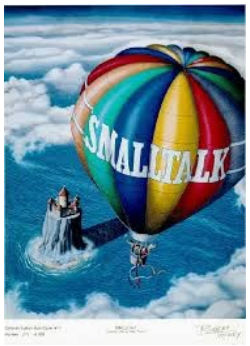
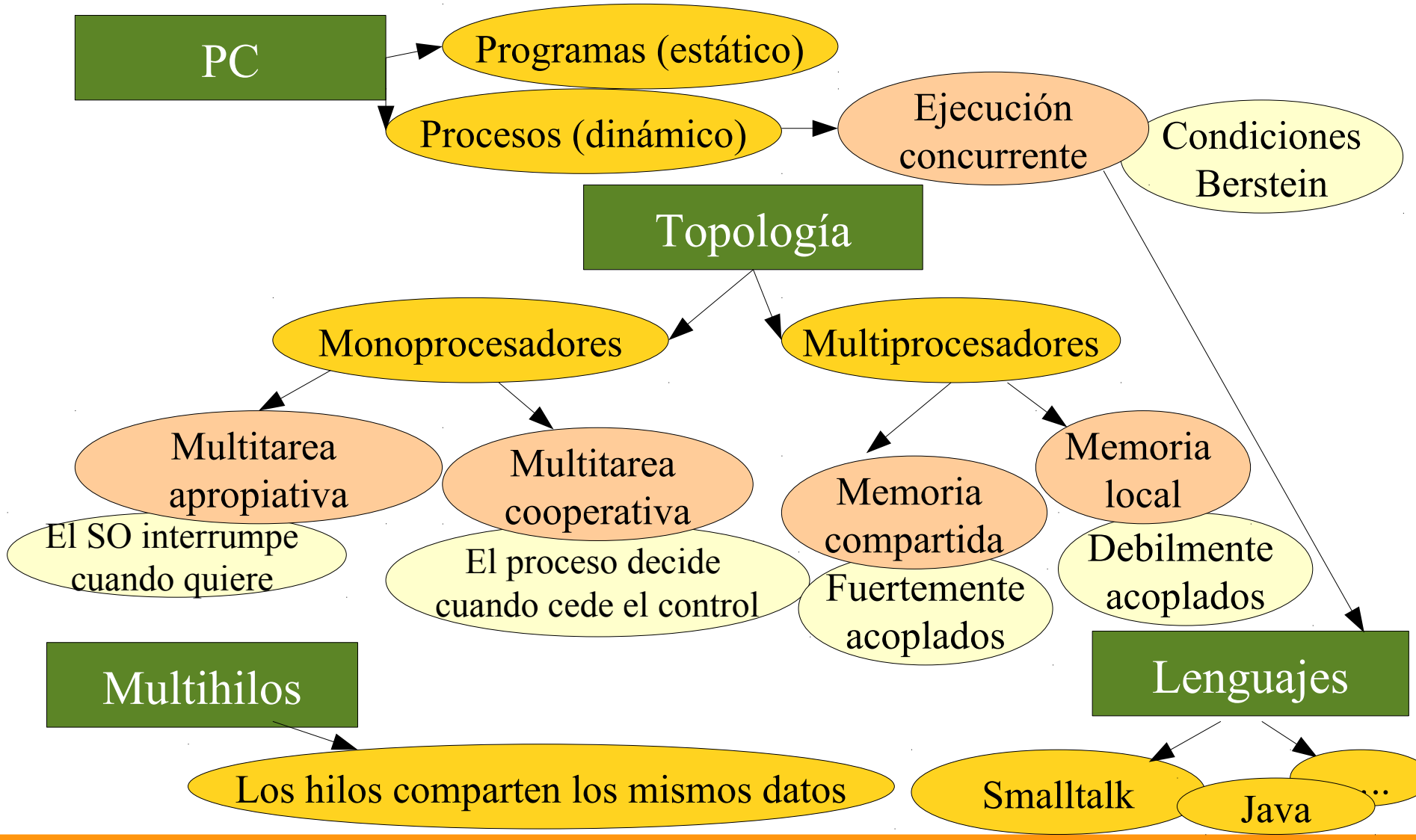
Departamento de Programación
Facultad de Informática
Universidad Nacional del Comahue

Programación Concurrente



Instrumentos de la concurrencia

Mapa conceptual



Ejecución concurrente

Las condiciones de Berstein determinan qué se puede ejecutar en forma concurrente

Sea

- $L(S_k)=\{a_1,a_2,...,a_n\}$ el conjunto de lectura del conjunto de instrucciones S_k formado por todas las variables cuyos valores son referenciados (se leen) durante la ejecución de las instrucciones en S_k
- $E(S_k)=\{b_1,b_2,...,b_n\}$ el conjunto de escritura del conjunto de instrucciones S_k formado por todas las variables cuyos valores son actualizados (se escriben) durante la ejecución de las instrucciones en S_k
- Para que dos conjuntos de instrucciones S_i y S_j se puedan ejecutar en forma concurrente, se debe cumplir que:

1.- $L(S_i) \cap L(S_j)= \emptyset$

2.- $E(S_i) \cap L(S_j)= \emptyset$

3.- $E(S_i) \cap E(S_j)= \emptyset$

Temario

- Repaso
- Correctitud
  - Propiedades de seguridad (safety)
  - Propiedades de viveza (liveness)
- Procesos
  - Thread (Java)
  - fork (Smalltalk)
- Sincronización
  - Bloqueo, Semáforos y Monitores

Smalltalk Concurrencia

Clase: BlockClosure

(categoría: scheduling)
métodos

fork, Crea y organiza el cuyo código del proceso corriendo en el receptor – corre en forma concurrente.

forkAndWait , Suspende el proceso actual y ejecuta el código en un nuevo proceso, cuando se complete el *resume* proceso actual.

forkAt: valorPrioridad , Crea y organiza el proceso en el receptor con una prioridad dada por valorPrioridad. Retorna el nuevo proceso creado.

Cómo crear procesos

```
[….“algunas sentencias”
Transcript show: 'Proceso']fork.
```

```
| bloqueAcciones proceso |
bloqueAcciones := [Transcript show: 'Proceso'].
proceso := bloqueAcciones fork.
```

Concurrencia en Smalltalk

Clase: BlockClosure

```
[….“algunas sentencias”
Transcript show: 'Proceso']fork.
```

crear procesos

```
| bloqueAcciones proceso |
bloqueAcciones := [Transcript show: 'Proceso'].
proceso := bloqueAcciones fork.
```

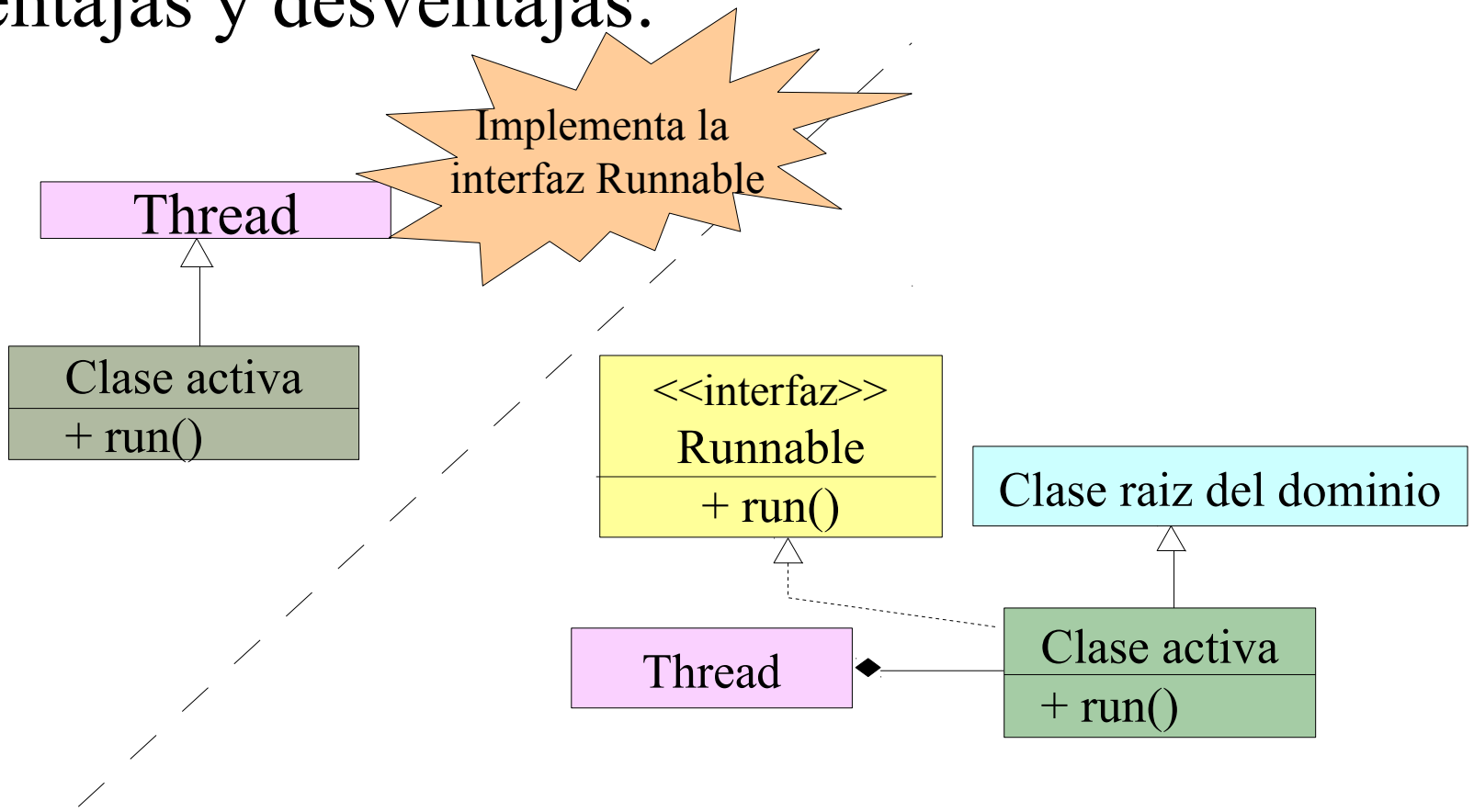
Probar....

```
[ i:=10.
  [i<500] whileTrue:[
          Transcript show: 'thread A'.
           i:= i+1]
]fork.
```

```
[ …..  'thread B'.
]fork.
```

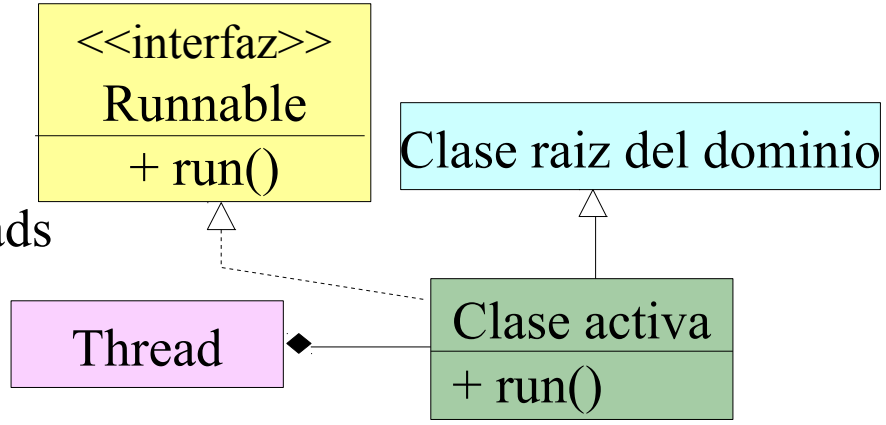
Cómo crear threads

- Ventajas y desventajas:



¿Por qué?

- Esta posibilidad crea un thread a través de la utilización de un objeto que implementa la interfaz Runnable, y con la que se incorpora el método run().
- De esta manera la clase MiClase debe implementar el método run().
- En el programa que declara el nuevo thread, se debe declarar primero el objeto t1 de la clase MiClase, y posteriormente cuando se crea el threads (se instancia el objeto t1 de la clase Thread) se le pasa como parámetro de su constructor.



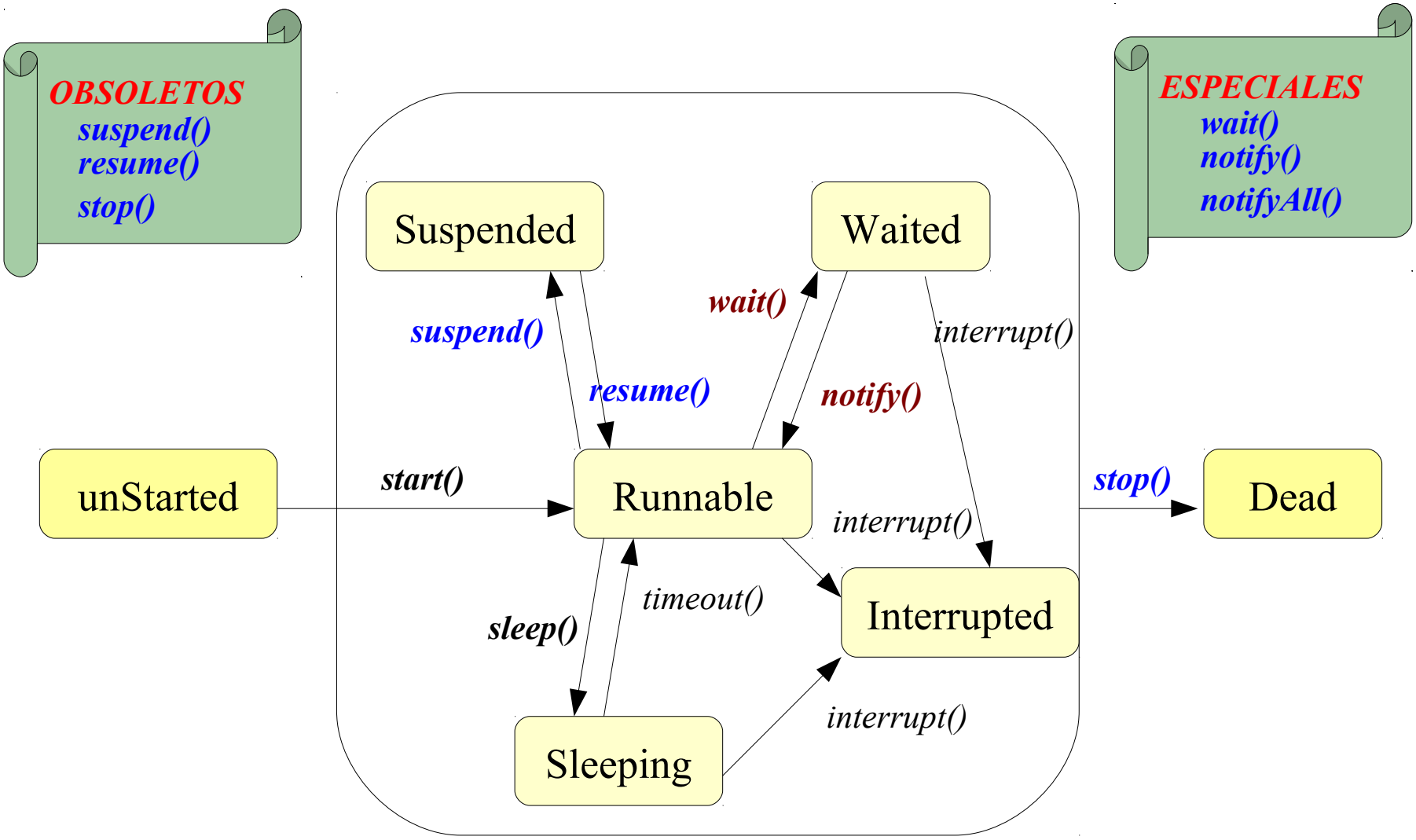
- Este es el procedimiento mas habitual de crear threads en java, ya que permite seudo-herencia múltiple (herencia y utilización de interfaz).

Java: Constructores clase Thread

- **Thread**()
- **Thread**(Runnable threadOb)
- **Thread**(Runnable threadOb, String threadName)
- **Thread**(String threadName)

- **Thread**(ThreadGroup groupOb, Runnable threadOb)
- **Thread**(ThreadGroup groupOb, Runnable threadOb, String threadName);
- **Thread**(ThreadGroup groupOb, String threadName)

Estados del Thread



Métodos de Thread

// muestran información del thread

int getPriority()

Thread.State getState()

void interrupt() – lo interrumpe

boolean isAlive()

boolean isDaemon()

boolean isInterrupted()

// sobreescriben información del thread

void setDaemon(boolean on) – Marca el thread como daemon o user thread

void setName(String name)

void setPriority(int newPriority)

// métodos específicos

void join() – espera para morir

void join(long ms) – espera ms milisegundos y muere.

void run() - principal, se reescribe en subclases

static void sleep(long millis) - dormir

void start() – comienza la ejecución.

static void yield() – Avisa al manejador (scheduler) que está dispuesto a usar un procesador

// obsoletos

void resume()

void stop()

void suspend()

// métodos heredados

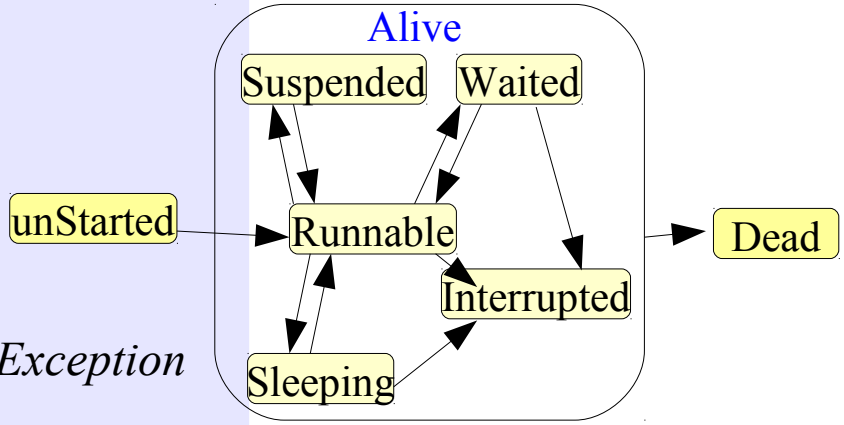
void notify() – levanta un thread que está esperando con wait

void notifyAll() - levanta todos los threads que esperaban

void wait() - hace que el corriente thread espere hasta que le llegue un notify

Repaso - Estados

- "UnStarted": El thread se instanció pero no comenzó
- "Alive": Se ejecuta el método run() .
- "Dead": El thread ha finalizado, por:
  - -Finaliza el *run()*
  - - Se invoca *stop()*. – En desuso
  - - No se recupera la excepción *InterruptedException* estando en Waiting o Sleeping.

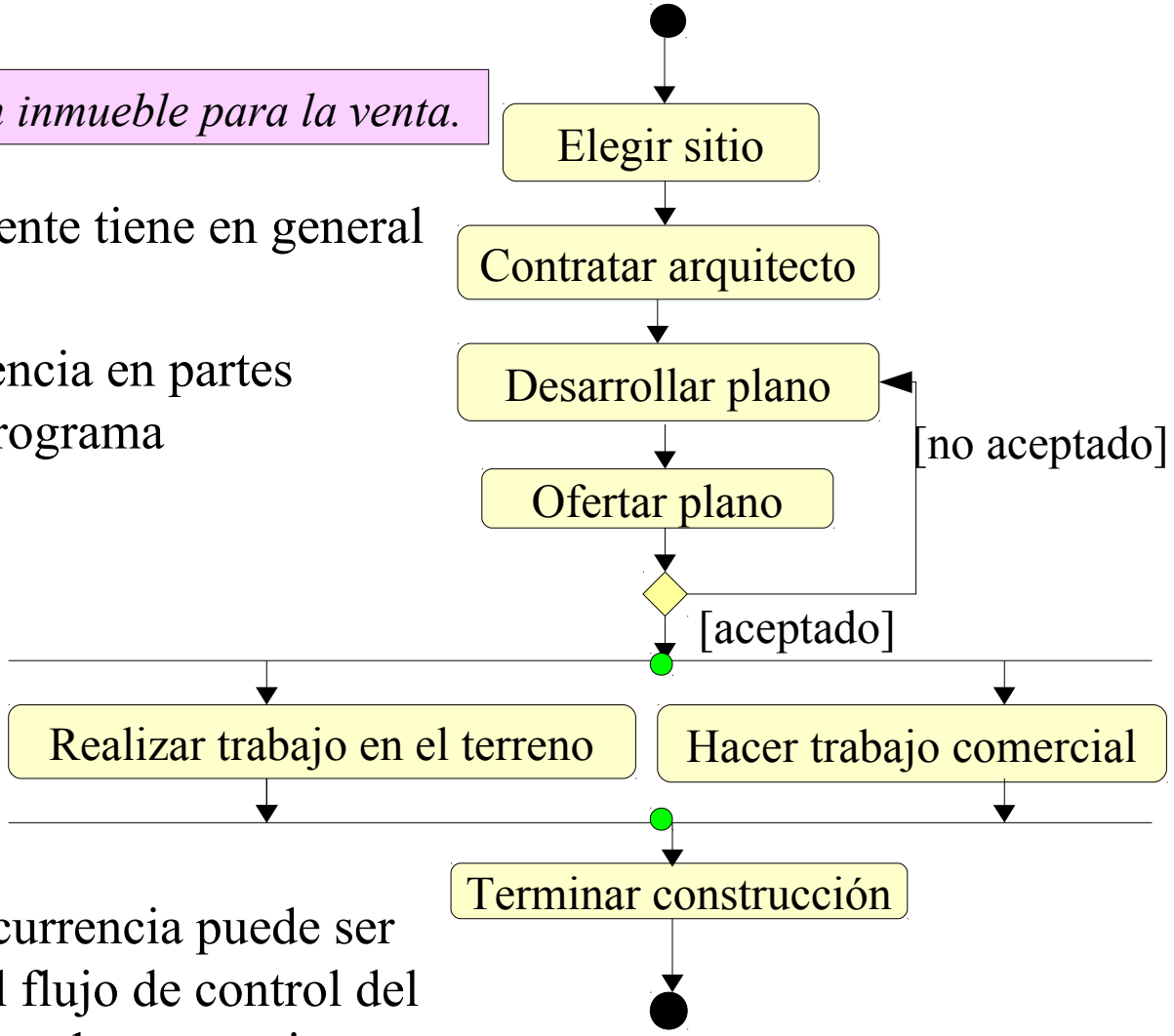


- "Runnable": ejecución planificada por el procesador.
- "Interrupted": se ha invocado el método Interrupt sobre el thread.
- "Suspended": En desuso
- "Sleeping": El thread está supendido temporalmente (durante el número de ms establecido por el argumento del método sleep() que lo ha dormido).
- "Waiting": El thread está supendido indefinidamente (sobre la variable de condición del objeto thread por haber invocado el mismo el método wait())

Problema

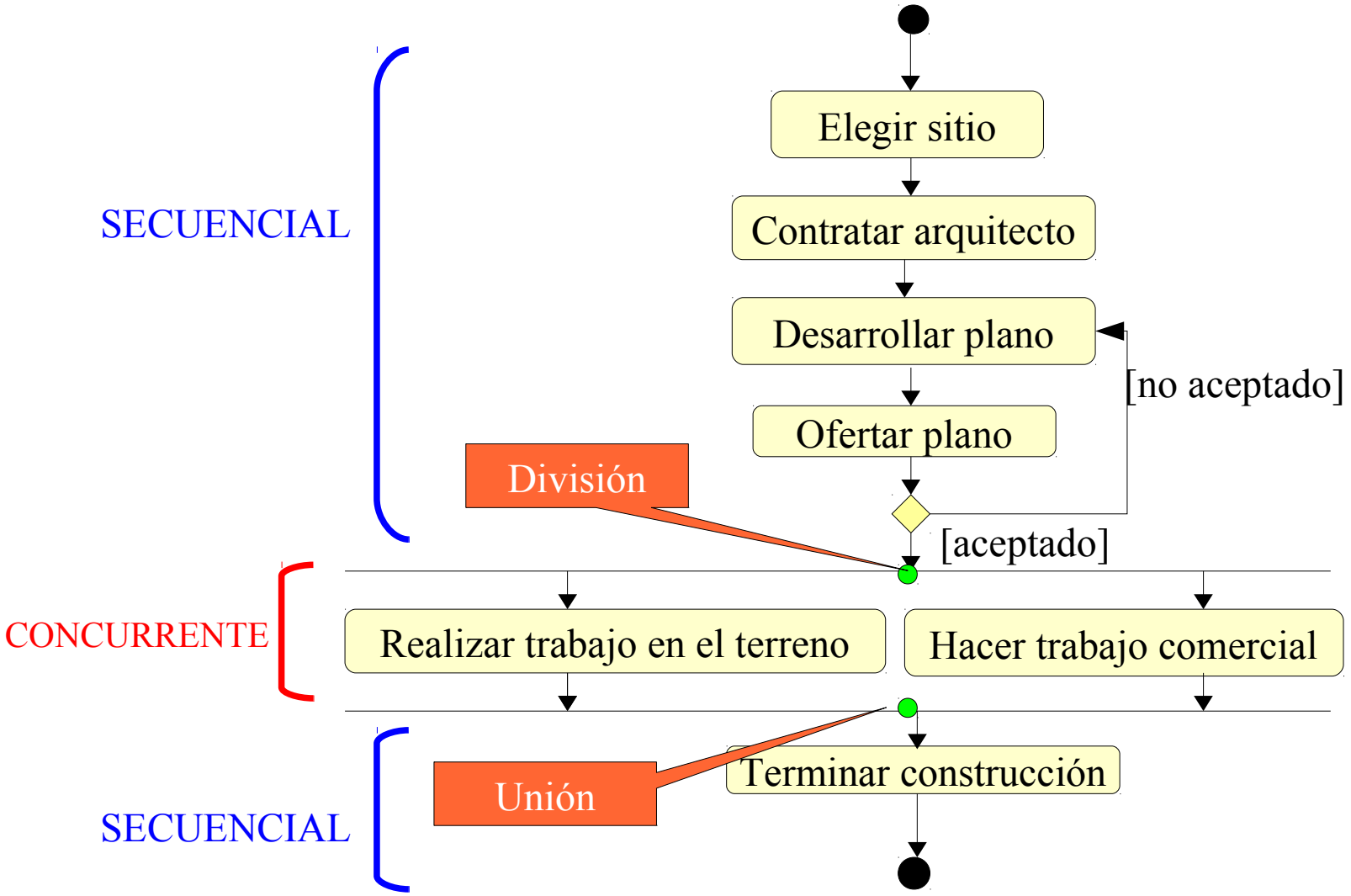
Ejemplo se desea construir un inmueble para la venta.

- Un programa concurrente tiene en general bloques secuenciales
  - Sólo hay concurrencia en partes concretas de un programa

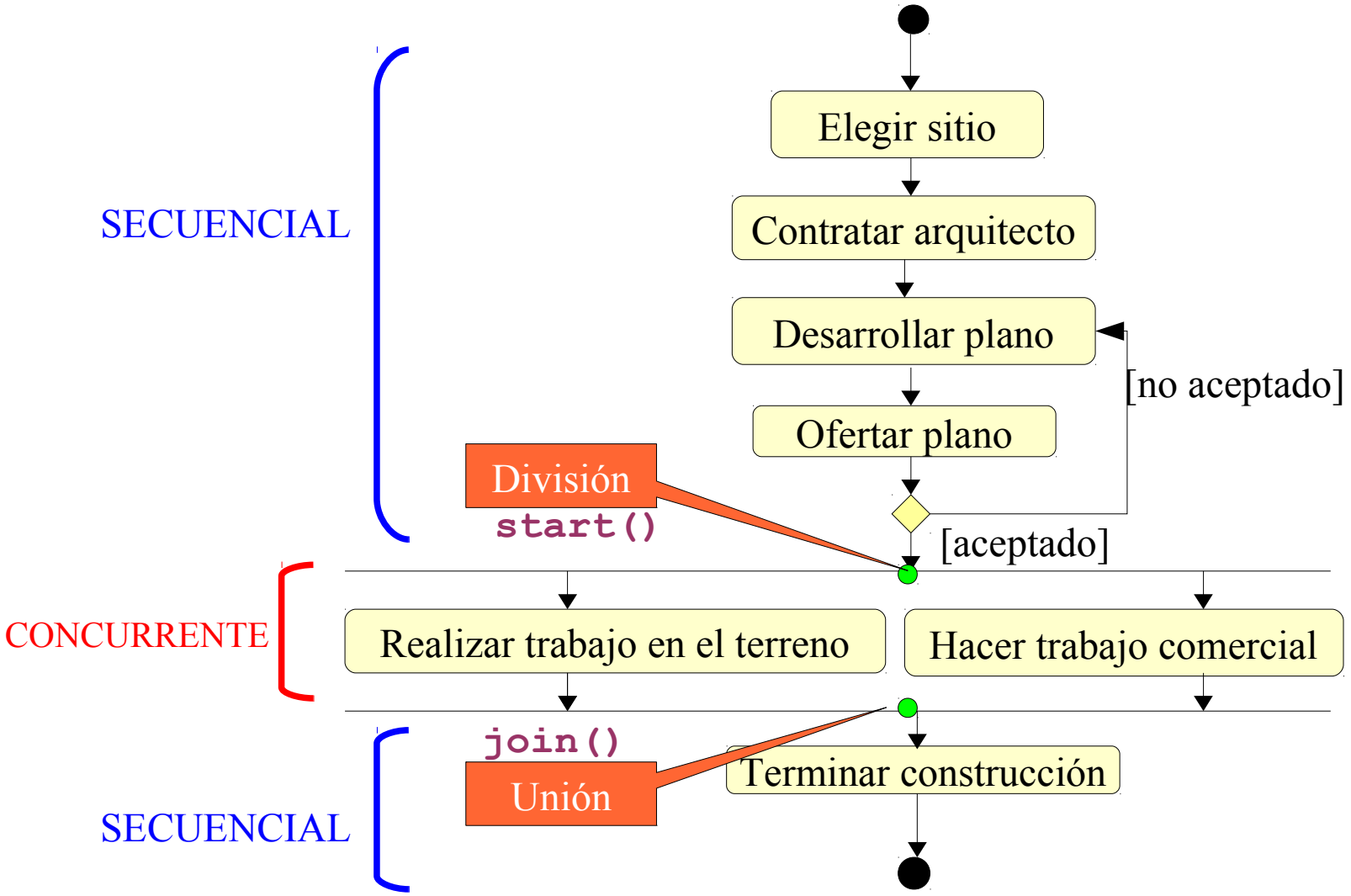


- Luego de utilizar concurrencia puede ser necesario reunificar el flujo de control del programa para continuar la secuencia

Reunificación de tareas



Reunificación de tareas: join



Java: Ejemplo join()

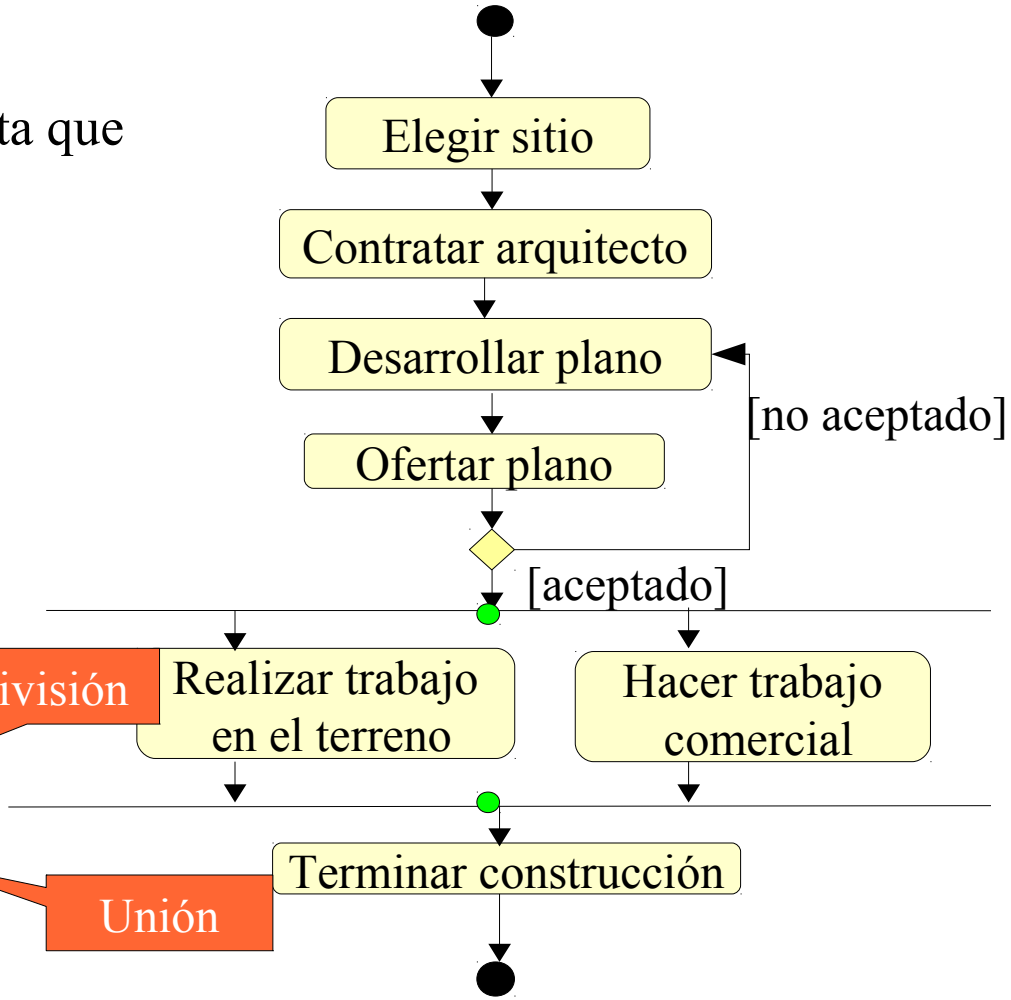
- **join()** el hilo activo espera hasta que muera (finalice) el run()

```
elegirSitio()
ContratarArquitecto()
REPETIR
        desarrollarPlano()
        ofertarPlano()
HASTA aceptado

trabajoTerreno.start()
trabajoComercial.start()
trabajoTerreno.join()
trabajoComercial.joint()

terminarConstruccion()
```

Concurrente

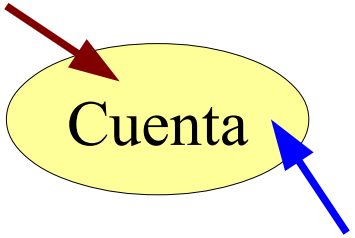
División

Unión



Problema tarjeta bancaria

- Un banco promociona la tarjeta joven, donde se abre una cuenta y se entregan 2 plasticos.



¿Pueden consultar saldos?

¿Pueden depositar?

¿Pueden extraer?

Condiciones para la concurrencia

Condiciones de Berstein → determinan el conjunto de instrucciones que se pueden ejecutar en forma concurrente y determinista

Se considera:

- El conjunto de instrucciones/procesos $S_1, S_2, \ldots S_n$
- El conjunto de lectura (variables accedidas)
- El conjunto de escritura (variables modificadas)

Cuenta
- saldo double
Cuenta (double saldoInicial) +getSaldo() : double +depositar(double imp): void +extraer (double imp): void

P1

P2

P3

$L(P1) = \{x\}$, $E(P1) = \{\}$

$L(P2) = \{x\}$, $E(P2) = \{x\}$

$L(P3) = \{x\}$, $E(P3) = \{x\}$

$L(S_i) \cap E(S_j) = \emptyset$

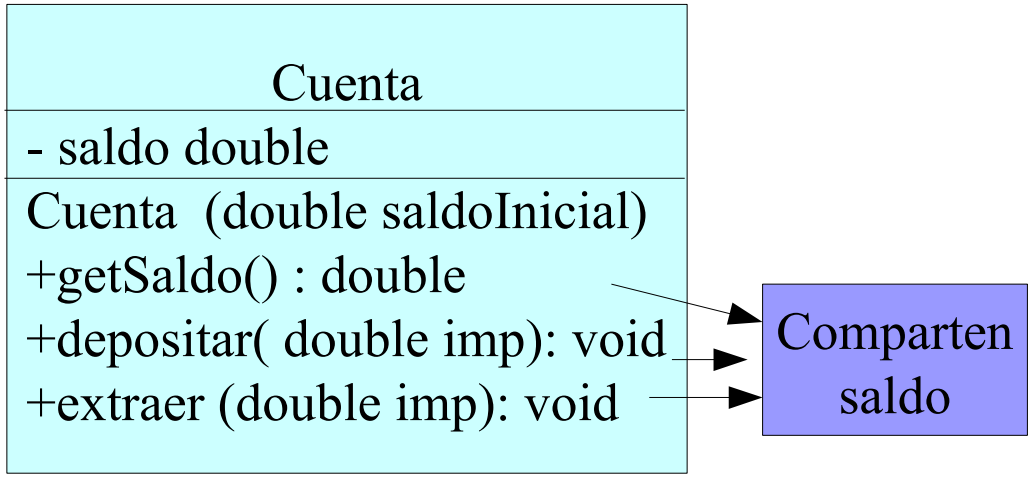
$E(S_i) \cap E(S_j) = \emptyset$

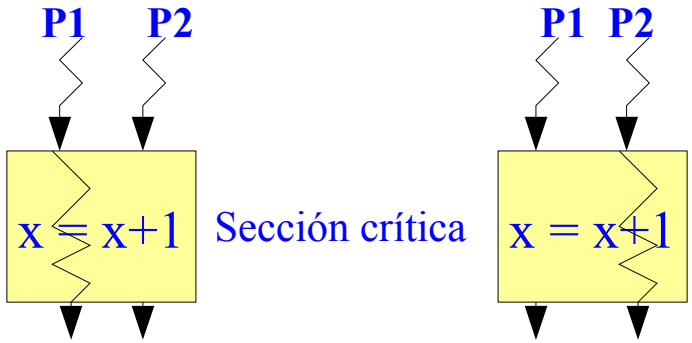
$E(S_i) \cap L(S_j) = \emptyset$

NO

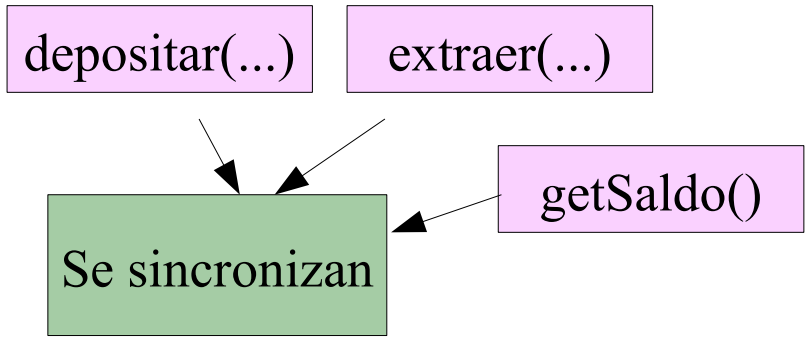
Sincronización

- ¿Cómo se sincroniza?





- Cada vez que un se deposita o se extrae en la cuenta cambia el saldo, sólo lo debe hacer una proceso por vez



Sincronización - Java

- Java utiliza Syncronized se utiliza para sincronizar objetos.
- El bloque synchronized lleva entre paréntesis la referencia a un objeto.
- Cada vez que un thread intenta acceder a un bloque sincronizado le pregunta a ese objeto si no hay algún otro thread ejecutando algun bloque sincronizado con ese objeto.
- Si otro thread ha realizado el bloqueo (lock), entonces el thread actual es suspendido y puesto en espera hasta que el lock se libere.
- Si el lock está libre, entonces el thread actual bloquea (lock) el objeto y entra a ejecutar el bloque.
- El lock se libera cuando el thread que lo tiene tomado sale del bloque por cualquier razón: termina la ejecución del bloque normalmente, ejecuta un return o lanza una excepción.
- El bloqueo es sobre un objeto en particular.
- Si hay dos bloques synchronized que hacen referencia a distintos objetos, la ejecución de estos bloques **no** será mutuamente excluyente.

Synchronized en Java

- Usar synchronized en un método de instancia es lo mismo que poner un bloque de synchronized(this){} que contenga todo el código del método.

  Es lo mismo:

```
public synchronized void metodo() {
   // codigo del metodo aca
}
```

```
public void metodo() {
   synchronized(this) {
      // codigo del metodo aca
   }
}
```

- Si el método es de clase entonces es lo mismo pero el bloque de synchronized se aplica a la clase.

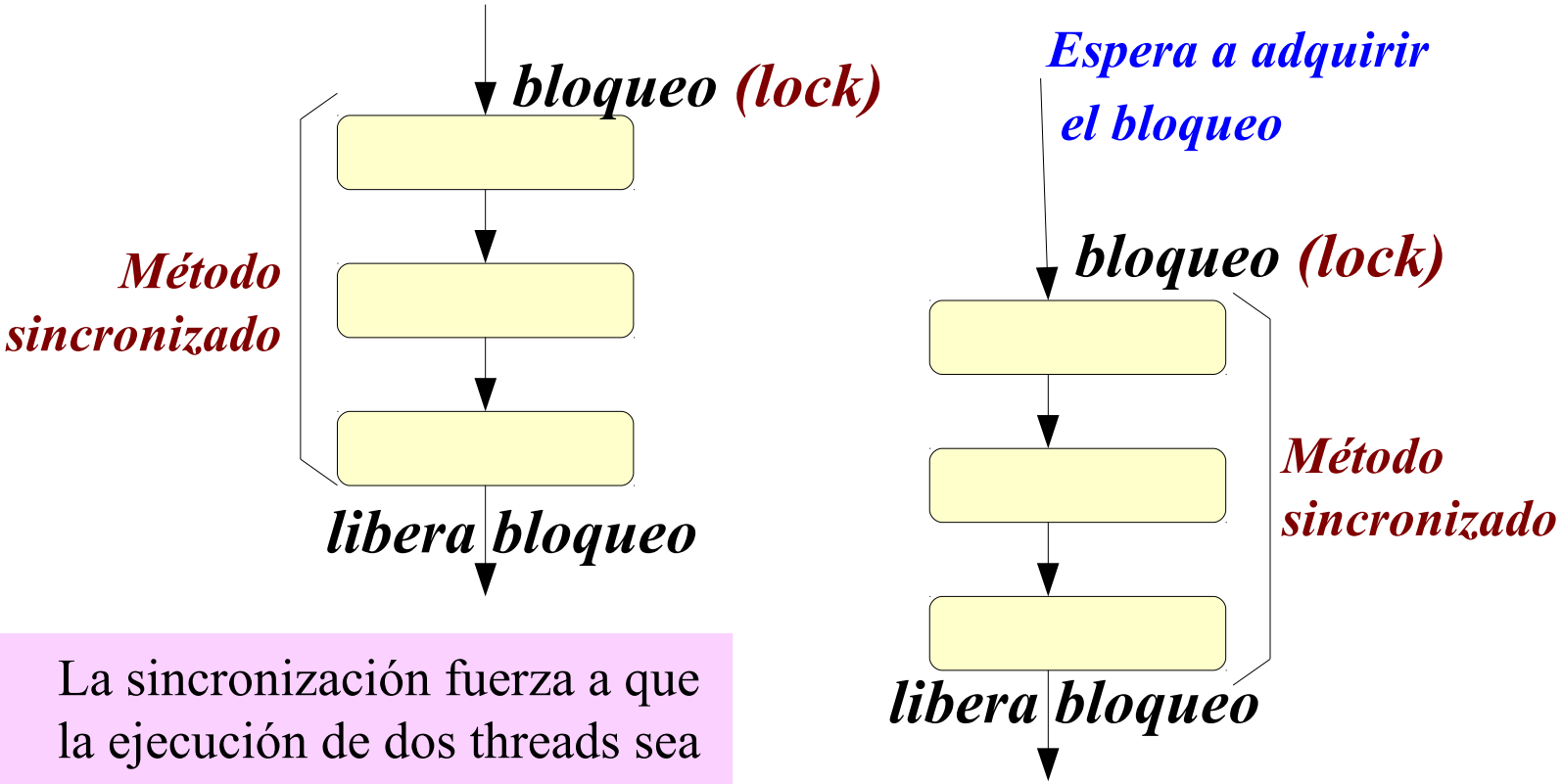
  Ejemplo, si el método está en la clase MiClase

```
public static synchronized void metodo() {
   // codigo del metodo aca
}
```

```
public static void metodo() {
   synchronized(MiClase.class) {
      // codigo del metodo aca
   }
}
```

Métodos sincronizados

- Para hacer una clase utilizable en ambientes de multitasking, ciertos métodos deben ser “sincronzados” .



La sincronización fuerza a que la ejecución de dos threads sea mutuamente excluyente.

Mecanismos de sincronización

- **Semáforos**: Dijkstra (1968).
  - Son componentes de muy bajo nivel de abstracción, de fácil comprensión y con una gran capacidad funcional.
  - Resultan peligrosos de manejar y son causa de muchos errores.
- **Secciones críticas** (Brinch Hansen, 1972)
  - Porción de código con variables compartidas y que se ejecutan en exclusión mutua.
  - Son componentes de los lenguajes de programación concurrente.
  - Alto nivel de abstracción, más fáciles y seguros de manejar.
- **Monitores** (Hoare 1974)
  - Módulos de programación de alto nivel de abstracción
  - Resuelven internamente, el acceso de forma segura a una variable o a un recurso compartido por múltiples procesos concurrentes.

Cerrojo

- Forma una sección crítica en cada proceso/hilo, desde que es tomado hasta que se libera.
- Como garantizan la exclusión mutua, muchas veces se los denomina mutex (por mutual exclusion).
- Restricciones de cerrojos:
  - Sólo el hilo dueño de un cerrojo puede desbloquearlo
  - La readquisición de un cerrojo no está permitida
  - Algo muy importante es que todos los procesos/hilos deben utilizar el mismo protocolo para bloquear y desbloquear los cerrojos en el acceso a los recursos
  - Si existe otro proceso que simplemente accede a los datos protegidos, no se garantiza la exclusión mutua
- Primitivas ***init(), lock() y unlock()***.

Exclusión Mutua

- **Cierres o cerrojos**:

debe entrar uno por vez.

  - Se utiliza cuando debe se comparten elementos por más de un hilo.
  - Cada proceso/hilo para tener acceso a un elemento compartido, deberá bloquear, con lo que se convierte en su dueño.
  - Al terminar de usar ese conjunto de elementos, el dueño debe desbloquear, para permitir que otro proceso/hilo pueda tomarlo a su vez.
  - Es posible que mientras un proceso/hilo esté accediendo a un recurso (siendo por lo tanto dueño del cerrojo), otro proceso/hilo intente acceder. En ese caso debe esperar hasta que el cerrojo se encuentre libre, para garantizar la exclusión mutua.
  - El proceso/hilo solicitante queda entonces en espera o pasa a estado de bloqueo según el algoritmo implementado.
  - Cuando el dueño del cerrojo lo desbloquea puede tomarlo alguno de los procesos/hilos que esperaban.

Semáforos en general

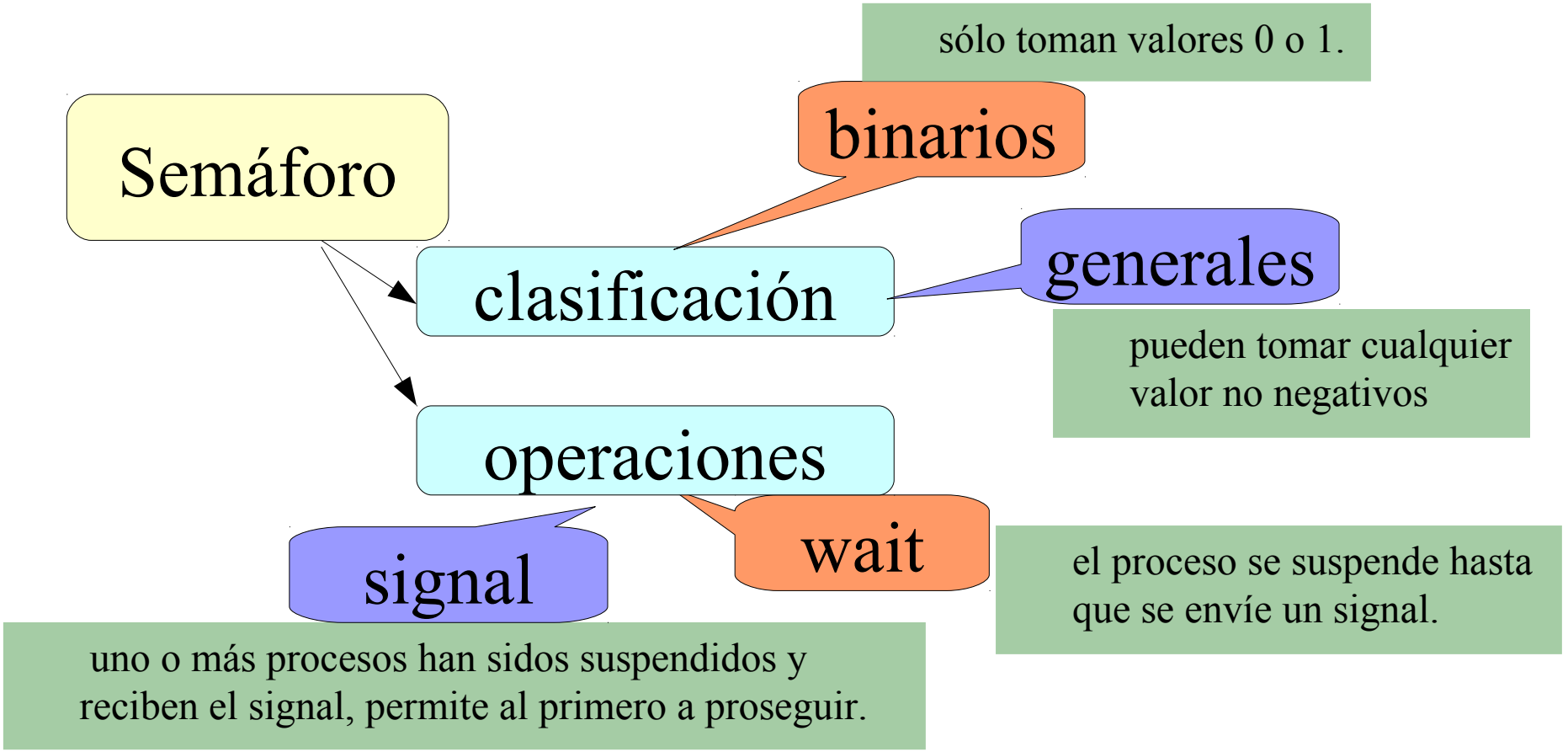
Tipo abstracto de datos que restringe o permite el acceso a recursos compartidos. (ej: recurso de almacenamiento del sistema)

- Se emplean para permitir el acceso a diferentes partes de programas (**secciones críticas**) donde se manipulan variables o recursos que deben ser accedidos de forma especial.

Semáforo binario: puede tomar solamente los valores 0 y 1.
Son usados cuando sólo un proceso puede acceder a un recurso a la vez.

- Según el valor con que son inicializados se permiten a más o menos procesos utilizar el recurso de forma simultánea.

Semáforos, generalidades



Semáforos, operaciones

- **Wait:**
  - Si el semáforo no es nulo (está abierto) decrementa en uno el valor del semáforo.
  - Si el valor del semáforo es nulo (está cerrado), el thread que lo ejecuta se suspende y se encola en la lista de procesos en espera del semáforo.
- **Signal:**
  - Si hay algún proceso en la lista de procesos del semáforo, activa uno de ellos para que ejecute la sentencia que sigue al wait que lo suspendió.
  - Si no hay procesos en espera en la lista incrementa en 1 el valor del semáforo.

Semáforos, en general

- Actúa como mediador entre un proceso y el entorno del mismo.
- Proporciona una forma simple de comunicación sincrónica.
- Garantiza que la operaciones de chequeo del valor del semáforo, y posterior actualización según proceda, sea siempre segura
- La inicialización del semáforo no es una operación segura por lo que no se debe ejecutar en concurrencia con otro proceso utilizando el mismo semáforo.
- Operaciones seguras:
  - Wait: El semáforo recibe el aviso de que necesita que se realice determinada actividad para continuar
  - El semáforo espera que del entorno se le envíe una señal **(signal)** para dar por confirmada la realización de la actividad, y enviarle el mensaje de *Resume* al proceso para que se reanude.

Semáforos, seudocódigo

- La ejecución de la operación signal(p) nunca provoca una suspención del thread que lo ejecuta.

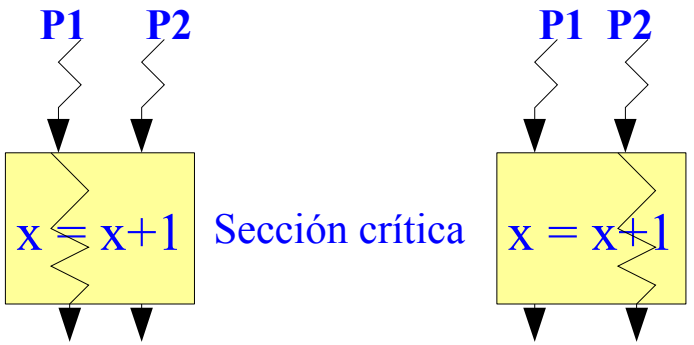
```
ALGORITMO wait
   SI semaforo>0 HACER
         semaforo ← semaforo -1
   SINO
     suspende proceso y lo pone en la cola del semáforo
   FIN SI
FIN ALGORITMO
```

- Si hay varios procesos en la lista del semáforo, la operación signal solo activa uno de ellos.
  - Este se elige de acuerdo con un criterio propio de la implementación (FIFO, LIFO, Prioridad, etc.).

```
ALGORITMO signal
   SI hay algún proceso en la cola
de semáforos HACER
         activa el primero de ellos
   SINO
         p ← p +1
   FIN SI
FIN ALGORITMO
```

Sincronización

- Sección crítica: porción de código con variables compartidas y que debe ejecutarse en exclusión mutua



- exclusión mutua (Mutex)

Algoritmo de exclusión mutua

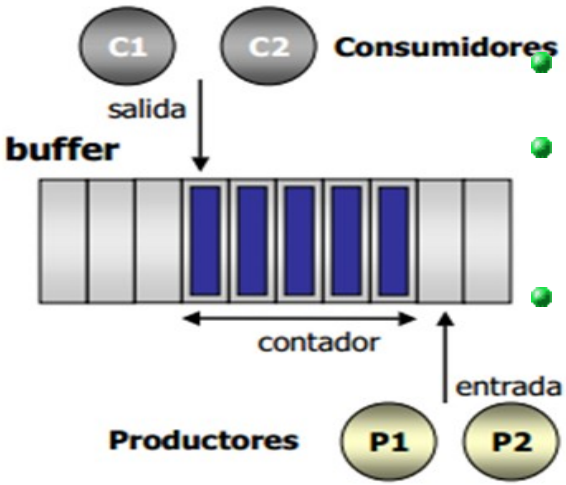
```
ALGORITMO exlusionMutua
        mutex: SemaforBinario
        p,q, r: Proceso

        inicial (mutex,1);
        cobegin p; q; r; coend;
FIN ALGORITMO
```

```
ALGORITMO proceso
    REPETIR
            wait(mutex)
            //código de la sección crítica
            signal(mutex)
    HASTA nunca
FIN ALGORITMO proceso
```

- Se plantea una solución del problema de exclusión mutua entre tres procesos ***p,q, r*** respecto de una sección crítica dada.
- Se utiliza un semáforo **"mutex"** para controlar el acceso a la misma.
  - Cuando toma el valor 0, algún proceso está en su sección crítica,
  - Cuando toma el valor 1 no hay ningún proceso ejecutándola.
  - El semáforo es inicializado a 1, ya que al comenzar, ningún proceso se encuentra en la zona crítica.

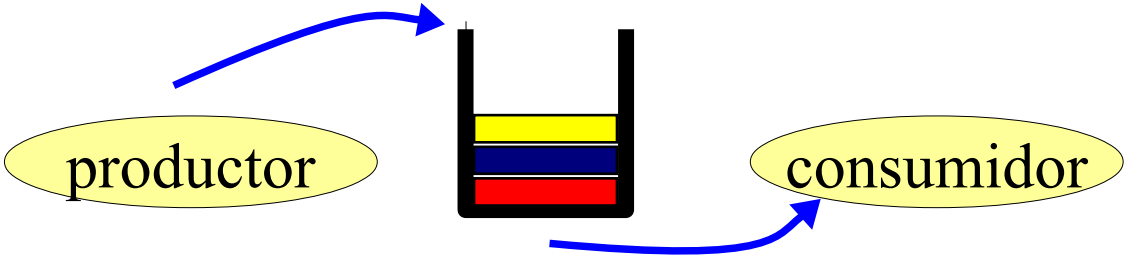
Productores/Consumidores

Supongamos una cola ilimitada



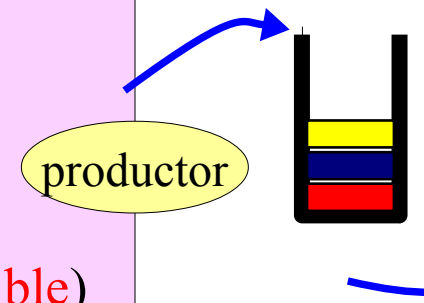
- El productor genera sus datos en cualquier momento
- El consumidor puede tomar un dato sólo cuando hay
- Para el intercambio de datos se usa una cola a la cual ambos tienen acceso, asi se garantiza el orden correcto
- Todo lo que se produce debe ser consumido

El consumidor no debe consumir más rápido de lo que produce el productor



Sincronización basada en semáforos

```
ALGORITMO productor
    REPETIR
        produce(dato)
        dejaDato(dato)
        signal(datoDisponible)
    HASTA nunca
FIN ALGORITMO
```




```
ALGORITMO consumidor
    REPETIR
        wait(datoDisponible)
        tomaDato(dato)
        consume(dato)
    HASTA nunca
FIN ALGORITMO
```

- El consumidor debe esperar hasta tanto que el dato que corresponda haya sido previamente colocado, tiene que esperar, luego debe contener una sentencia wait.
- Esta versión resuelve el problema productor consumidor entre dos procesos pero no la posibilidad que existan varios productores y varios consumidores
- Para el intercambio de datos se usa una cola a la cual ambos tienen acceso, asi se garantiza el orden correcto