

# Concurrencia

Paradigmas de la programación  
FaMAF-UNC 2015  
basado en filminas de Tom Mitchell  
capítulo 15

# concurrency

dos o más secuencias de eventos ocurren en paralelo

- Multiprogramación
  - una computadora ejecuta varios programas a la vez
  - cada programa funciona secuencialmente
  - las acciones de un programa pueden suceder entre dos pasos de otro
- Multiprocesos
  - dos o más procesadores conectados
  - los programas de un procesador se comunican con los del otro
  - las acciones pueden suceder en simultáneo

proceso: programa secuencial ejecutándose en un procesador

# la promesa de la concurrencia

- velocidad
  - si una tarea toma  $t$  tiempo en un procesador, no debería tomar  $t/n$  tiempo en  $n$  procesadores?
- disponibilidad
  - si un proceso está ocupado, otro podrá ayudar
- distribución
  - los procesadores de diferentes lugares pueden colaborar en solucionar un problema o trabajar juntos
- si los humanos lo hacen, las computadoras también
  - visión, cognición, parecen actividades fuertemente paralelas

# retos

- los programas concurrentes son más difíciles de sacar bien
- algunos problemas son inherentemente secuenciales, y requieren mucha coordinación y comunicación entre sub-problemas si los paralelizamos
- problemas a resolver
  - comunicación – cómo recibir o enviar información
  - sincronización – cómo esperar a otro proceso
  - atomicidad – garantizar que no se va a parar en la mitad y dejar todo embrollado

# y en paradigmas...

- cómo hacen los lenguajes de programación para facilitar la programación concurrente y la corrección de los programas concurrentes?

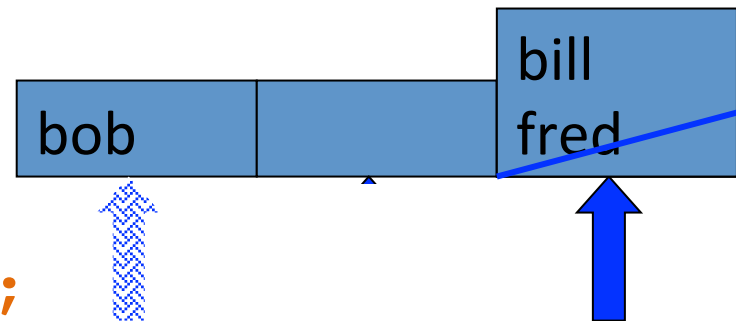
# qué esperamos de un lenguaje concurrente: construcciones

- hilos (*threads*) como valores de expresiones
  - se pueden pasar hilos a funciones, crear hilos como resultado de una llamada a función
- abstracciones de comunicación
  - comunicación síncrona
  - canales con búfer asíncronos que preservan el orden de los mensajes
- control de concurrencia
  - exclusión mútua
  - alguna forma de cerrojo (*lock*)
  - la atomicidad no es tan común en los lenguajes

# problema básico: condiciones de carrera

```
procedure anotar(persona)  
begin  
    numero := numero + 1;  
    list[numero] := persona;  
end;
```

- problema con la ejecución en paralelo



```
anotar(fred) | anotar(bill);
```

# resolver conflictos entre procesos

- sección crítica
  - dos procesos que acceden a un recurso compartido
  - comportamiento inconsistente si se intercalan dos acciones

permitimos un solo proceso en la sección crítica

- deadlock
  - un proceso puede estar reteniendo locks mientras está esperando a otros
  - el *deadlock* ocurre cuando ningún proceso puede continuar



# locks y espera

<initialize control de concurrencia>

Thread 1:

<wait>

anotar(fred); // sección crítica

<signal>

Thread 2:

<wait>

anotar(bill); // sección crítica

<signal>

se necesitan operaciones atómicas  
para implementar wait

# primitivas de exclusión mútua

- test-and-set atómico
  - una instrucción lee y escribe de forma atómica
  - es una instrucción de hardware común
  - se combina con el bucle ocupado-en espera para implementar mutex
- semáforo
  - se evita el bucle ocupado-en espera
  - se mantiene una cola de procesos en espera
  - el scheduler tiene acceso al semáforo, el proceso duerme
  - se inhabilitan interrupciones durante las operaciones de semáforo

# estado de la cuestión

- la programación concurrente es difícil
  - difícil darse cuenta de cuándo se van a dar condiciones de carrera o deadlocks
- los lenguajes deberían ayudar con patrones, abstracciones, paradigmas útiles
- se necesitan otras herramientas
  - el testing es difícil por la explosión combinatoria
  - hay detectores de condiciones de carrera
    - estáticos: conservadores, quizás demasiado restrictivos
    - en ejecución: más prácticos por el momento

# Cobegin/coend

- primitiva de concurrencia limitada

```
x := 0;
```

```
cobegin
```

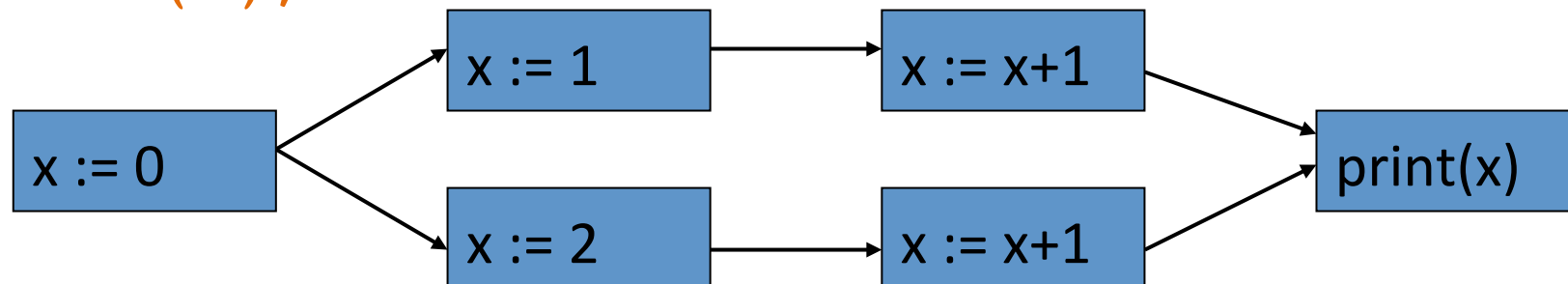
```
begin x := 1; x := x+1 end;
```

```
begin x := 2; x := x+1 end;
```

} ejecutar bloques  
secuenciales en paralelo

```
coend;
```

```
print(x);
```



la atomicidad está al nivel de la sentencia de asignación

# propiedades de cobegin/coend

- ventajas
  - crea procesos concurrentes
  - comunicación mediante variables compartidas
- limitaciones
  - sin exclusión mútua
  - sin atomicidad
  - el número de procesos está fijo por la estructura del programa
  - no se pueden abortar procesos, todos se tienen que completar para que el padre pueda seguir

Concurrent Pascal, P. Brinch Hansen, Caltech, 1970' s

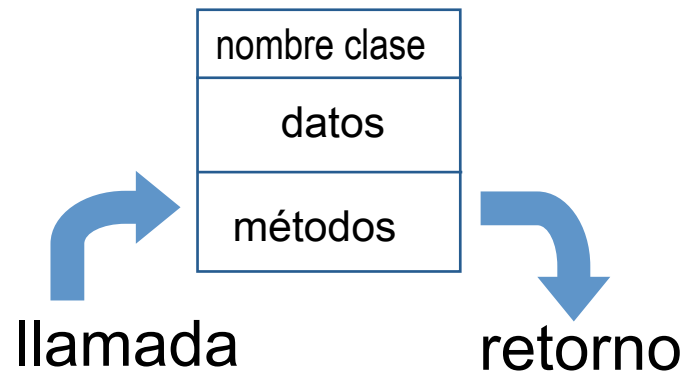
# Actores

[Hewitt, Agha, Tokoro, Yonezawa, ...]

- cada actor (objeto) tiene un script
- en respuesta a un input, el actor atómicamente
  - crea nuevos actores
  - inicia una comunicación
  - cambia su estado interno
- la comunicación es
  - en búfer, sin pérdida
  - no se garantiza el orden de llegada
    - preservar el orden es más difícil de implementar
    - el programador puede reconstruir el orden si es necesario
    - es ineficiente tener comunicación ordenada si no es necesario

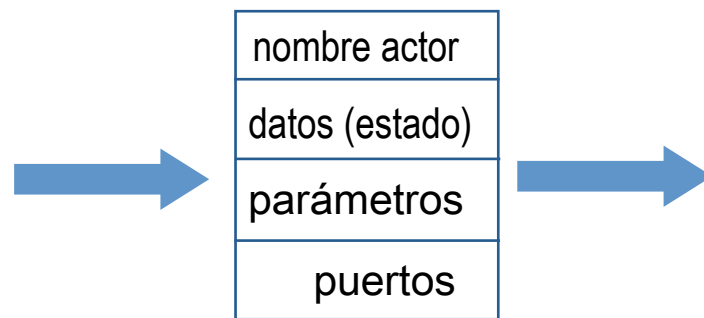
# programas orientados a actores

orientación a objetos:



lo que fluye a través  
de un objeto es  
control secuencial

orientación a actores:

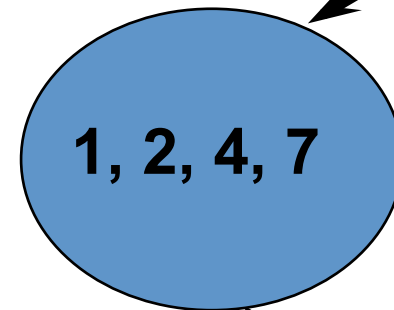
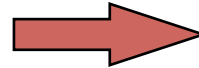
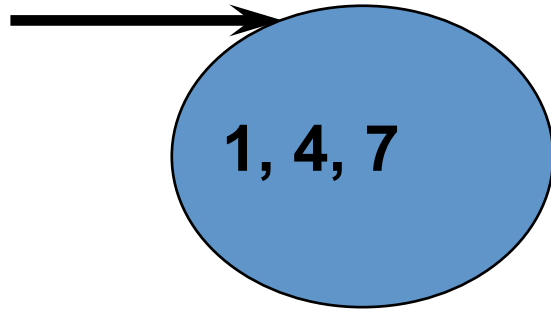


lo que fluye a través  
de un objeto son  
streams de datos

input data    output data

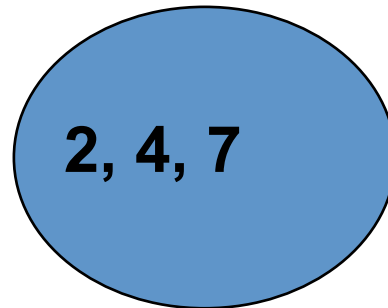
# ejemplo

insertar 2



mínimo

1





# pila implementada con actores

parámetros

**nodo\_pila** con **acquaintances** **contenido** y **link**

if operación requerida es **pop** y **contenido** != nil then

convertirse en **forwarder** hacia **link**

enviar **contenido** al cliente

if operación requerida es **push(nuevo\_contenido)** then

let **P**=new **nodo\_pila** con **acquaintances** actuales (**clon**)

convertirse en **nodo\_pila** con **acquaintances**

**nuevo\_contenido** y **P**

difícil de leer pero tiene la semántica

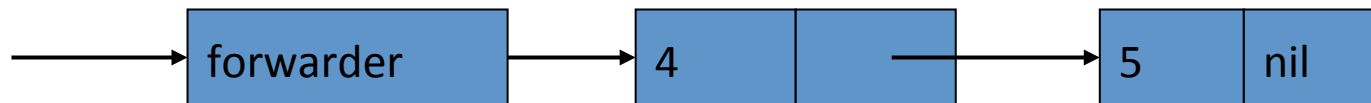
denotacional de una pila, sólo que *forwarder*  
es inusual....

# *Forwarder*

pila antes de *pop*



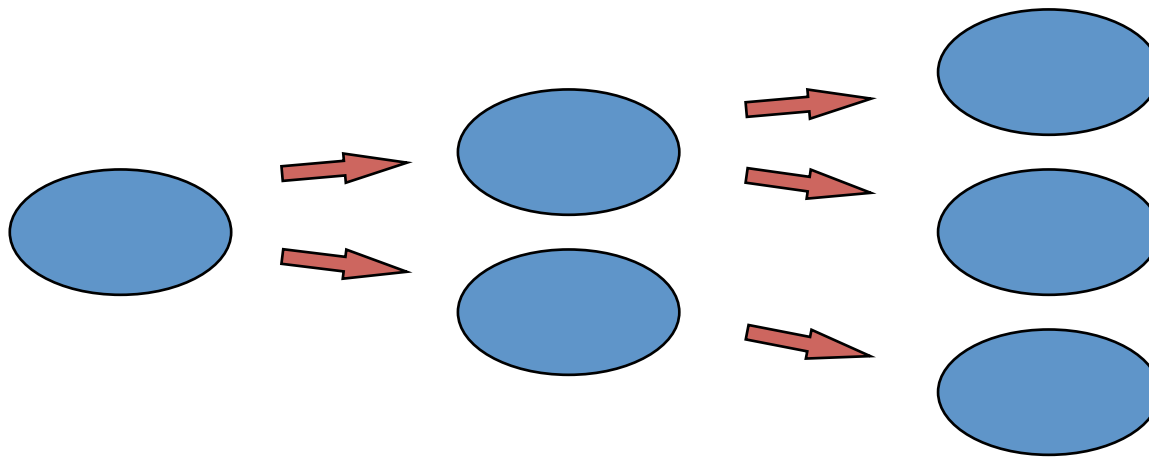
pila después de *pop*



el nodo “desaparece” convirtiéndose en un nodo *forwarder*.  
El sistema maneja los nodos forwardeados de forma que se convierten invisibles al programa.

# concurrency

muchos actores pueden operar concurrentemente



- no es necesario controlar explícitamente la concurrencia
- los mensajes enviados por un actor pueden ser recibidos y procesados por otros secuencialmente o concurrentemente

# Pros y contras del modelo de actores

- lenguaje de programación de alto nivel
  - comunicación por mensajes
  - exclusión mútua: si se envían dos mensajes, los actores reaccionan atómicamente al primero que reciben antes de ver el segundo
  - la concurrencia es implícita
- quizás demasiado abstracto para algunas situaciones?
  - cómo hacer fork de varios procesos para hacer cómputo especulativo, y terminarlos a todos cuando uno encuentra la solución?
    - el coordinador de todo el resto de actores se convierte en cuello de botella

# ML concurrente [Reppy, Gansner, ...]

- threads
  - son una entidad (un tipo)
- comunicación
  - canales sincrónicos
- sincronización
  - canales
  - eventos
- atomicidad
  - no hay soporte específico en el lenguaje

# concepto previo a Java: monitor

- acceso sincronizado a datos privados
- encapsula:
  - datos privados
  - conjunto de procedimientos (métodos)
  - política de sincronización
    - máximo un objeto puede ejecutar un procedimiento del monitor a la vez, se dice que el proceso está **en** el monitor
    - si un proceso está en el monitor, los otros procesos que llaman a un procedimiento del monitor serán demorados
- en terminología moderna: objeto sincronizado

# concurrency en Java

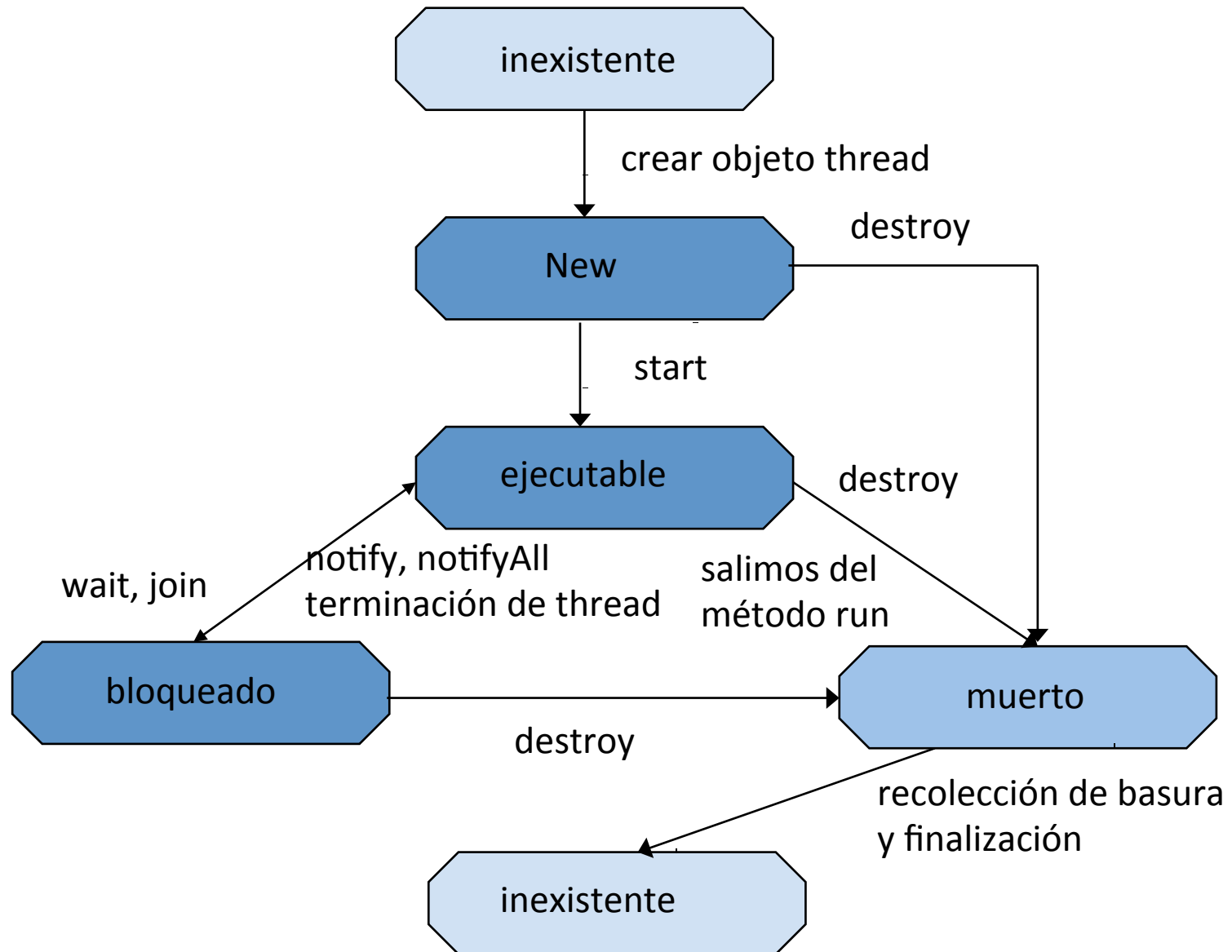
- threads
  - crean un proceso creando un objeto thread o implementando una interfaz
- comunicación
  - variables compartidas
  - llamadas a métodos
- exclusión mútua y sincronización
  - cada objeto tiene un lock (heredado de la clase Object)
    - métodos y bloques sincronizados
  - operaciones de sincronización (heredado de la clase Object)
    - `wait` : pausar el thread actual hasta que le hacen `notify`
    - `notify`: despertar threads que están esperando

# threads de Java

- un thread es un conjunto de instrucciones que se ejecutan secuencialmente
- en Java, son objetos
  - de la clase `Thread`
  - métodos que se heredan de `Thread`:
    - `start` : reproduce (*spawn*) un nuevo thread, hace que la máquina virtual llame al método `run`
    - `suspend` : congela la ejecución
    - `interrupt` : congela la ejecución y lanza una excepción
    - `stop` : fuerza el thread a detenerse



# estados de un thread en Java



## un problema con la especificación de Java

- Java permite acceso a objetos parciales

```
class Broken {  
    private long x;  
    Broken() {  
        new Thread() {  
            public void run() { x = -1; }  
        }.start();  
        x = 0;  
    }  
}
```

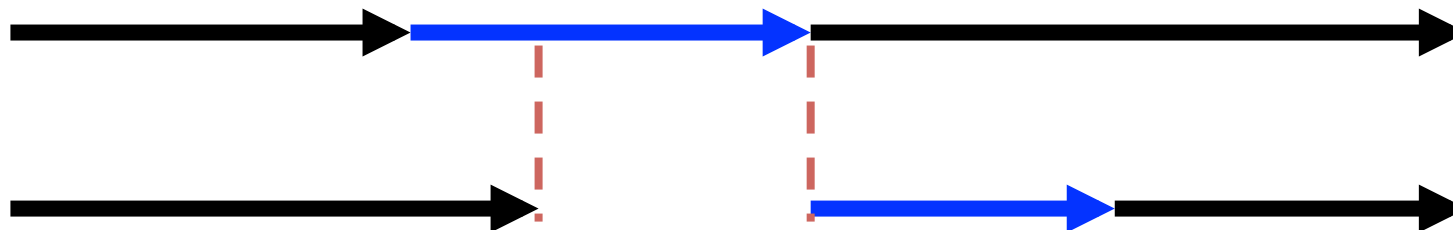
Thread created within constructor can access the object not fully constructed

# interacción entre threads

- variables compartidas
  - dos threads pueden asignar/leer a la misma variable
  - responsabilidad del programador: hay que evitar condiciones de carrera explícitamente
- llamadas a métodos
  - dos threads pueden llamar a métodos del mismo objeto, y ahí se ponen en funcionamiento los locks propios del objeto
  - cada objeto tiene un lock interno, heredado de Object
  - las primitivas de sincronización se basan en ese lock

# sincronización

- provee exclusión mútua
  - dos threads pueden tener acceso al mismo objeto
  - si un thread llama un método sincronizado, el objeto se cierra (*lock*)
  - si otro thread llama a un método sincronizado del mismo objeto, el thread se bloquea hasta que el objeto se abre (*unlock*)



# métodos sincronizados

- se marcan con una palabra clave

```
public synchronized void commitTransaction(...) {...}
```
- máximo un método sincronizado puede estar activo
- los métodos no sincronizados se pueden llamar: el programador tiene que ir con cuidado!
- no es parte de la signatura del método
  - un método sincronizado es equivalente a uno no sincronizado con el mismo cuerpo
  - una subclase puede sustituir un método sincronizado con uno no sincronizado

# ejemplo

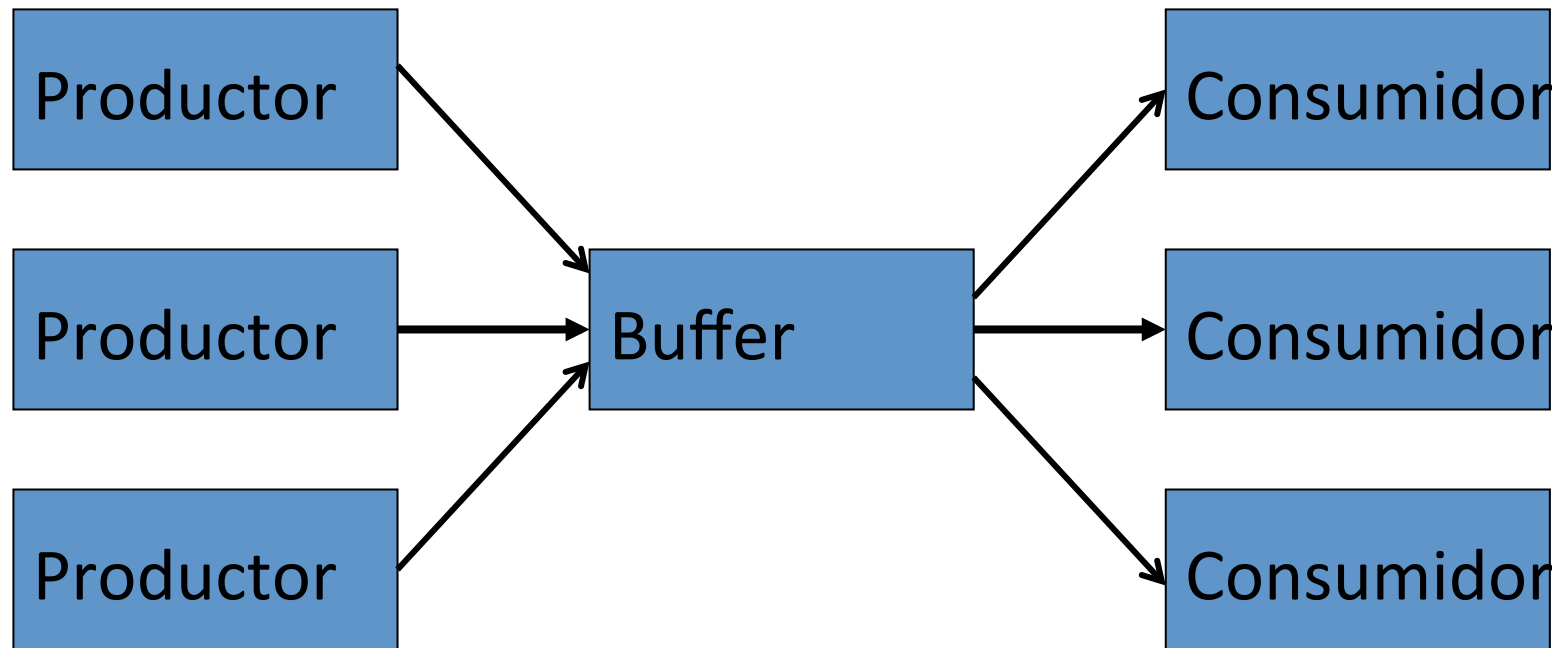
```
class LinkedCell {    // una celda con un valor y un link a la siguiente
    protected double value;
    protected final LinkedCell next;
    public LinkedCell (double v, LinkedCell t) {
        value = v; next = t;
    }
    public synchronized double getValue() {
        return value;
    }
    public synchronized void setValue(double v) {
        value = v;    // la asignación no es atómica
    }
    public LinkedCell next() {    // no requiere sincronización
        return next;
    }
}
```

# Join, otra forma de sincronización

espera a que termine un thread

```
class Future extends Thread {  
    private int result;  
    public void run() { result = f(...); }  
    public int getResult() { return result;}  
}  
...  
Future t = new future;  
t.start() // empieza un nuevo thread  
...  
t.join(); x = t.getResult(); // espera y obtiene  
resultado
```

# productor-consumidor?



- la llamada a métodos es síncrona
- cómo se hace en Java?



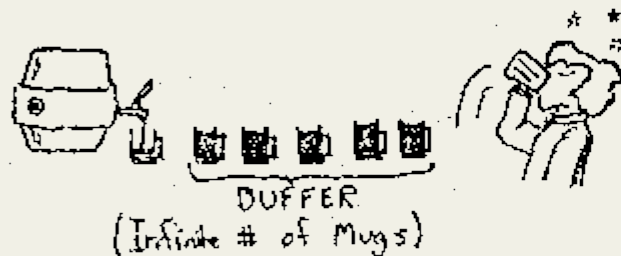
# A GRAPHIC EXAMPLE OF THE PRODUCER/CONSUMER PROBLEM

Michael Vignaro

① PRODUCER CONSUMER

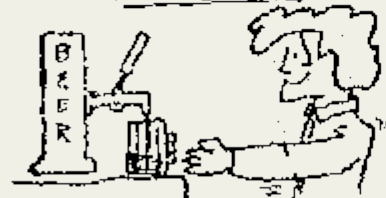


②



③

PROBLEM -

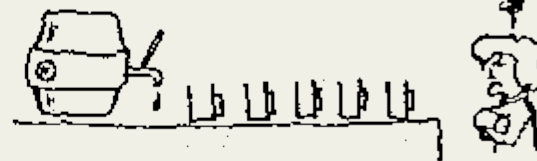


Consumer takes from buffer before producer is done adding to it - trouble!  
This is solved by \_\_\_\_\_

(fill in the blank)

④

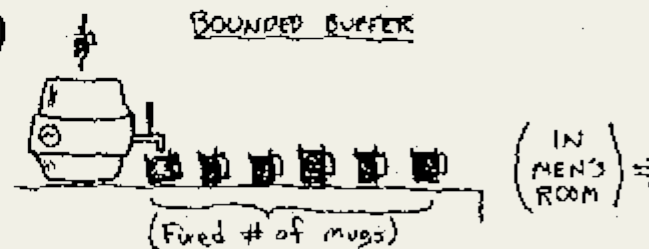
One-way balcony



The consumer must wait for producer to produce before it can consume...

⑤

BOUNDED BUFFER



If the consumer is busy (can't consume), the producer must wait, if the buffer is full, for the consumer to start consuming again. The processes are now \_\_\_\_\_  
(fill in)


# solución al productor-consumidor

- no se puede resolver solamente con locks: se usan los métodos `wait` y `notify`
- el consumidor espera hasta que hay algo en el buffer
  - mientras espera, tiene que dormir, con el método `wait`
  - se requiere un bucle de rechequeo de la condición que lo tiene durmiendo
- el productor informa a los consumidores que esperan cuando hay algo en el buffer
  - tiene que despertar por lo menos a un consumidor, con el método `notify`

# Stack<T>: métodos produce, consume

```
public synchronized void produce (T object) {  
    stack.add(object); notify();  
}
```

```
public synchronized T consume () {  
    while (stack.isEmpty()) {  
        try {  
            wait();  
        } catch (InterruptedException e) { }  
    }  
    Int lastElement = stack.size() - 1;  
    T object = stack.get(lastElement);  
    stack.remove(lastElement);  
    return object; }  
}
```



por qué  
este loop?

# Concurrent garbage collector

- How much concurrencia?
  - Need to stop thread while mark and sweep
  - Other GC: may not need to stop all programa threads
- Problem
  - programa thread may change objects during collection
- Solution
  - Prevento read/write to memory area
  - Details are subtle; generational, copying GC
    - Modern GC distinguishes short-lived from long-lived objects
    - Copying allows read to old area if writes are blocked ...
    - Relatively efficient methods for read barrier, write barrier

# limitaciones de las primitivas de Java 1.4

- no se puede volver del intento de adquirir un lock
  - no se puede desistir después de haber esperado por un cierto tiempo
  - no se puede cancelar un intento de adquirir un lock después de una interrupción
- no se puede alterar la semántica de un lock
  - protección de lectura vs. escritura, fairness, ...
- no hay control de acceso a sincronización
  - cualquier método puede hacer `synchronized(obj)` para cualquier objeto
- la sincronización se hace con métodos y bloques
  - no se puede adquirir un lock en un método y soltarlo en otro