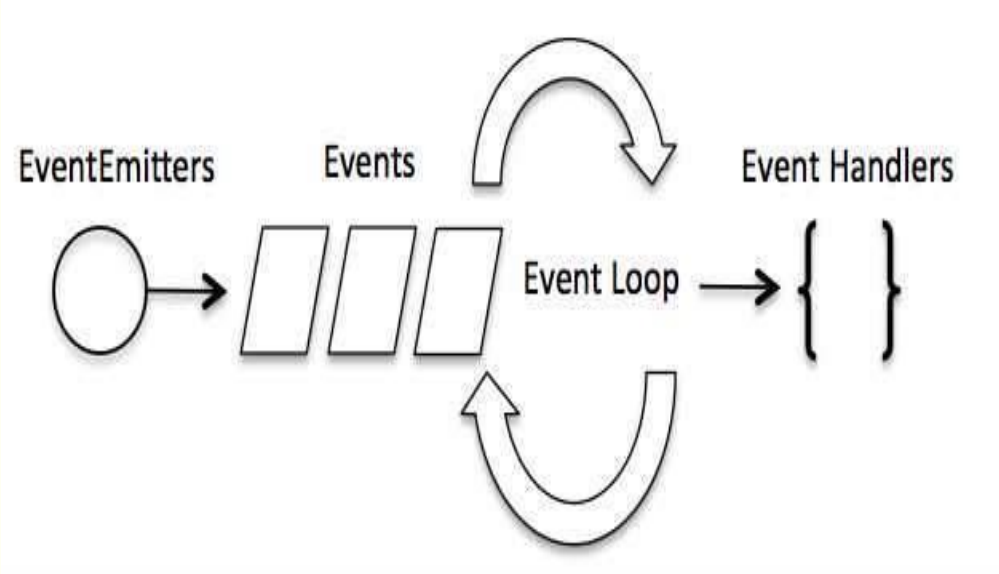
PROGRAMACIÓN ORIENTADA A EVENTOS

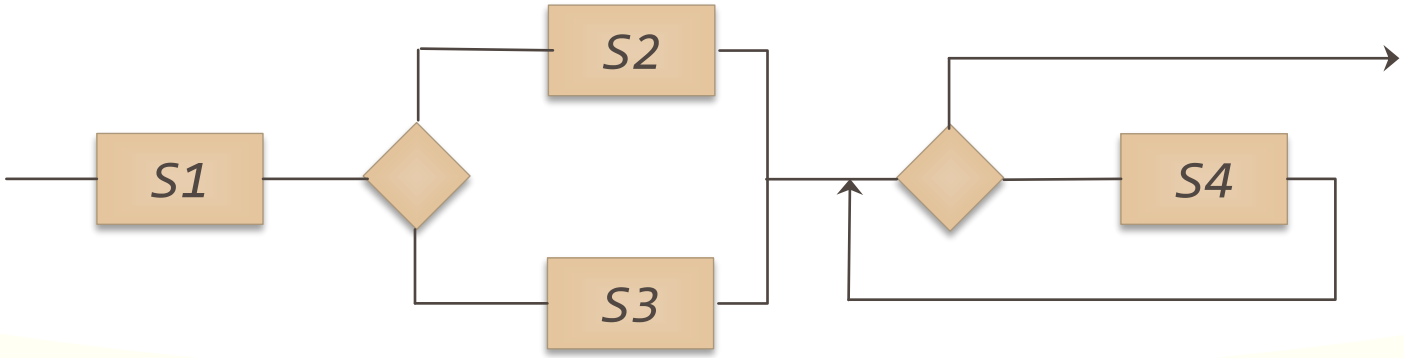


Índice

- Introducción
- Sistemas basados en eventos (SBE)
  - Características de un SBE
- El paradigma de POE
- Diseño de programas orientados a eventos
- Infraestructura de eventos
- Concurrencia

Programación Secuencial

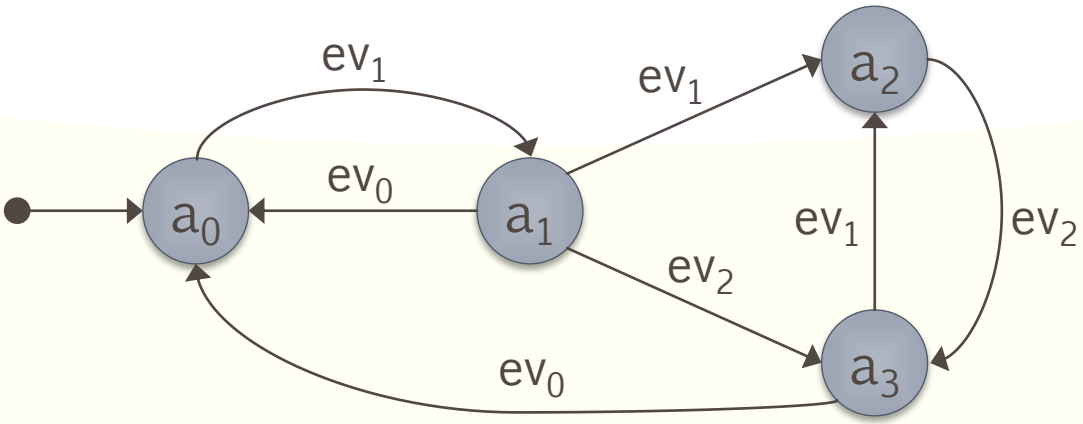
- Flujo de ejecución
  - Está formado por la secuencia de sentencias que componen el programa.



- Estado del programa
  - Definido por el punto del programa en que se encuentra la ejecución (valor de los datos en éste)

Programación Orientada a Eventos (1)

- Programación guiada por sucesos que ocurren (eventos) y que, dependiendo de cual suceda, ejecutan un código u otro (habitualmente una función)
  - Ejemplo típico, las GUI
- No existe un único flujo de ejecución



Programación Orientada a Eventos (2)

- Estado en un POE
  - Más difícil de determinar que en un programa secuencial
    - Habitualmente son programas concurrentes (multihilo)
    - Tienen distintos módulos que se ejecutan a distintas velocidades
    - Los eventos ocurren en cualquier momento y, a menudo, dependen de la interacción del usuario
    - Los módulos se ejecutan de forma simultánea, algunos puede que nunca se ejecuten y de otros pueden existir varias instancias en ejecución
    - Los módulos pueden compartir información (datos)

Programación Orientada a Eventos (3)

- Eventos
  - Un **evento** es una *ocurrencia observable*
    - *Ocurrencia* porque sucede en algún momento
    - *Observable* porque es posible que un observador note que sucedió
  - Tipos de eventos
    - *Externos*
      - Pulsación de una tecla o botón del ratón
    - *Internos*
      - Vencimiento de un temporizador
      - Datos en líneas de comunicaciones

Programación Orientada a Eventos (4)

- Respondiendo a eventos
  - Si un observador se interesa por un evento, puede responder a éste de alguna forma
  - En programación cuando un observador responde a un evento, se dice que *maneja* o *controla* éste y que el observador es el *manejador* o *controlador de evento*

Programación Orientada a Eventos (5)

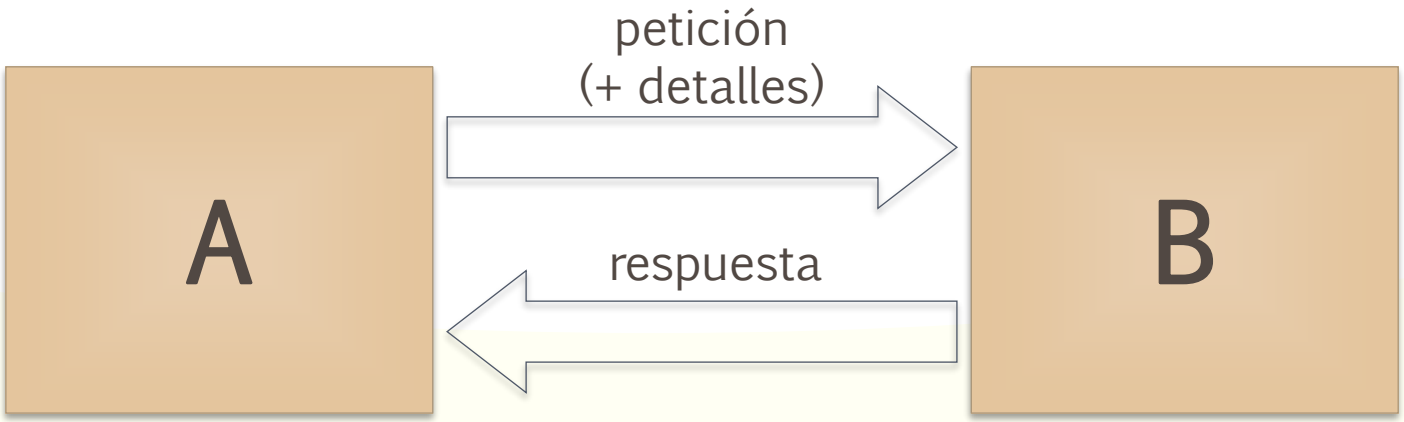
- Fuentes de eventos
  - Algunos eventos ocurren porque algún agente fue responsable de provocarlos
  - En programación, se dice que el agente que causa el evento es *la fuente del evento* y que ésta *dispara o lanza el evento*
    - Cuando ocurre un evento, pero no estamos interesados en su origen, se suele decir informalmente que el *evento se dispara* (aunque éstos por si mismos no son agentes capaces de actuar)

Sistemas Basados en Eventos (1)

- Sistema
  - Conjunto de agentes sujetos a un conjunto definido de comportamientos e interacciones
    - En el sistema, el comportamiento de un agente en cualquier momento depende únicamente de su *estado* y éstos pueden interactuar entre sí de varias formas.
      - Se describirán tres tipos de interacción basados en eventos

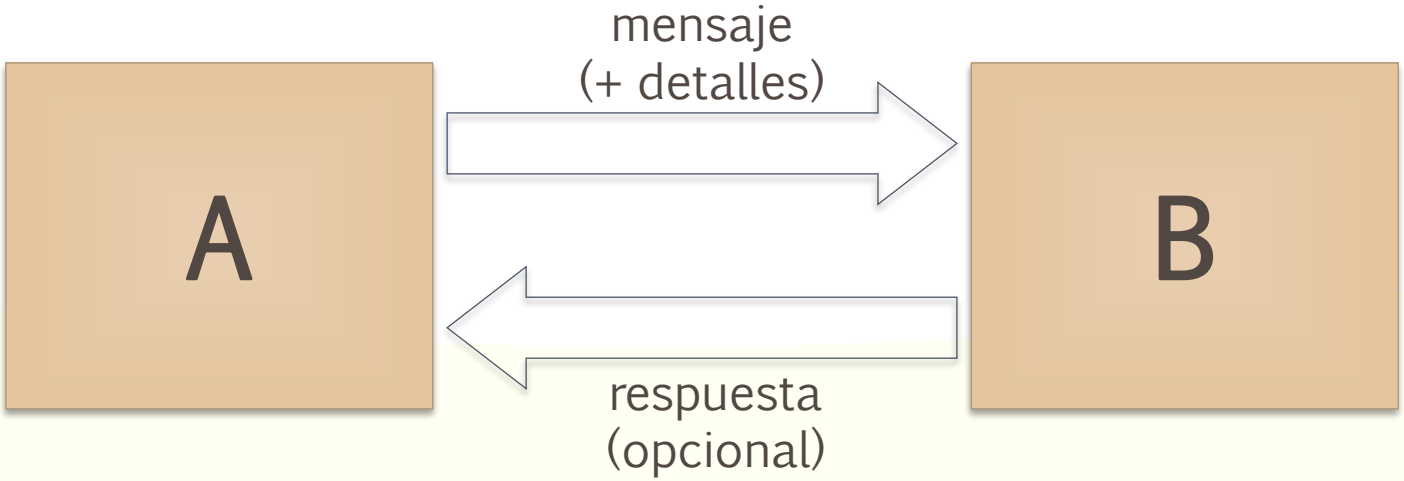
Sistemas Basados en Eventos (2)

- Petición-Respuesta
  - Una interacción *petición-respuesta* se encuentra entre dos agentes



Sistemas Basados en Eventos (3)

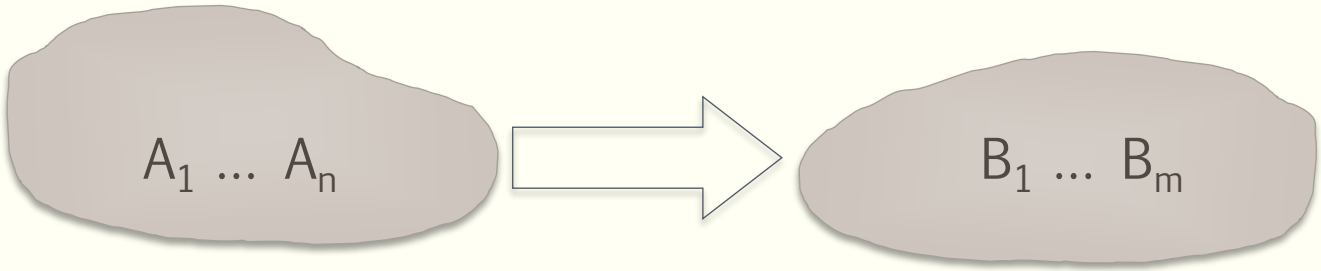
- Paso de mensaje
  - Una interacción de *paso de mensaje* también se encuentra entre dos agentes



  - A diferencia del caso previo el agente B no está obligado a responder

Sistemas Basados en Eventos (4)

- Publicación-Subscripción
  - Una interacción *publicación-subscripción* implica múltiples agentes.
    - Los agentes B_1 a B_m se suscriben a un servicio de mensajes que indica que desean recibir cierto tipo de mensajes
    - Los agentes A_1 a A_n publican varios tipos de mensajes para el servicio. Si el agente Ai publica un mensaje que le interesa al agente B_j, éste recibirá una copia del mensaje.



Sistemas Basados en Eventos (5)

- Estado del sistema
  - El *estado de un sistema* es la descripción completa del sistema en un momento dado.
    - La actividad de los agentes del sistema que interactúan con otros agentes mediante eventos, modifica el estado del sistema
- Sistema basado en eventos (SBE)
  - Un *sistema basado en eventos* es un sistema en el que las interacciones entre agentes se rigen por eventos.

Sistemas Basados en Eventos (6)

- Sistemas discretos y eventos
  - Un sistema es discreto si cada estado del sistema se puede describir mediante una cantidad finita de memoria y si, en cualquier intervalo de tiempo finito, el estado del sistema cambia un número finito de veces.
    - Para cada estado de un sistema discreto, siempre hay un próximo estado.
  - Un sistema de eventos discreto, es un sistema discreto basado en eventos

Sistemas Basados en Eventos (7)

- Características
  - Basado en estado
    - Estado de datos
    - Estado de control
  - No determinista
  - Acoplamiento débil
  - Control descentralizado

El Paradigma de POE (1)

- La POE es un paradigma.
  - Es una forma de pensar acerca de los problemas y sus soluciones
  - Proporciona abstracciones
    - El **modelo de evento** es su *abstracción principal*.

El Paradigma de POE (2)

- El modelo de evento
  - El núcleo del paradigma POE está en el concepto de evento
  - Hay tres tipos de objetos computacionales asociados con cada evento:
    - *la fuente del evento*
    - *el objeto del evento*
    - *uno o más controladores o manejadores de evento*

El Paradigma de POE (3)

- Fuentes de eventos
  - La *fuente del evento* es el creador del evento.
    - Decimos que la fuente del evento activa un evento cuando crea un objeto de evento y lo prepara para los manejadores
- Objetos de evento
  - Un objeto de evento encapsula los datos críticos asociados con el evento
- Manejadores de eventos
  - Los controladores de eventos responden a eventos llevando a cabo las acciones especificadas por el programador

El Paradigma de POE (4)

- Eventos versus invocaciones de método
  - La idea que hay detrás de ambas técnicas es similar: provocar la ejecución de cierto código, pero hay varias diferencias importantes.
    1. La fuente del evento y el manejador de eventos están mucho más débilmente acoplados que los objetos en una relación estándar de llamador y llamado

El paradigma de POE (5)

2. Los controladores de eventos están registrados con las fuentes de eventos en tiempo de ejecución
   - Es posible conectar diferentes controladores con la misma fuente de eventos en diferentes momentos durante la ejecución. Lo que es bastante diferente de la semántica de tiempo de compilación y enlace de las llamadas a métodos tradicionales
3. Puede haber cero, uno o múltiples controladores registrados para un evento.
   - Enviar un evento a varios controladores, también conocido como multidifusión, resulta útil cuando hay varias vistas que dependen de los mismos datos.

El paradigma de POE (6)

4. Los controladores de eventos no devuelven ninguna información al origen del evento.
   - Por la naturaleza del paradigma, los manejadores de eventos tienen un tipo de devolución `void`.
5. Múltiples fuentes de eventos pueden disparar eventos al mismo controlador.
   - Este tipo de multiplexación se ve con frecuencia en GUI cuando hay múltiples formas de realizar la misma tarea.
6. En la mayoría de los lenguajes basados en eventos, el origen del evento no se bloquea esperando a que los manejadores se completen.
   - Dispara el evento, luego continúa funcionando; es decir, la fuente del evento y el controlador de eventos se ejecutan de forma asíncrona.

El paradigma de POE (7)

7. Puede haber un retraso entre el momento en que se desata el evento y cuando cada controlador lo procesa.
    - Este retraso puede ser causado por una acumulación de otros eventos que están a la espera del procesamiento, otros controladores que se ejecutan para el evento actual, un retraso de la red si el controlador está en un sistema remoto u otras muchas razones.

El paradigma de POE (8)

Al igual que con cualquier paradigma, los lenguajes y las bibliotecas que implementan el soporte para los sistemas basados en eventos varían.

Cada una de las propiedades enumeradas anteriormente puede estar o no soportada

Diseño de POEs (1)

- Respuesta ante eventos
  - Programas de algoritmo de respuesta único
    - La respuesta ante un evento se obtiene mediante un algoritmo único
    - Pueden ser con memoria o sin memoria
      - Si no tienen memoria la respuesta ante un evento es siempre la misma

evento(datos)	Acción
ponerColorFondo(Color RGB)	Color de Fondo ← RGB

Diseño de POEs (2)

- Si tienen memoria la respuesta ante un evento puede variar

evento(datos)	Acción
ponerColorFondo(Color RGBA)	Color de Fondo ← RGBA
Opacidad(Real a)	Color de Fondo ← RGBa

- Programas de respuesta según estado
  - La respuesta varía según el estado global; es decir, según los eventos que se hayan producido previamente
  - Se pueden tener algoritmos de respuesta distintos según el estado

Diseño de POEs (3)

- Se diferencian según el mecanismo para mantener el estado
  - **Basados en grafos**
  - Basados en máquina de pila
- Para completar la respuesta que se da ante un evento se requiere indicar para cada evento
  - La respuesta para cada estado
  - La acción inicial

- Ejemplo
  - Un cronómetro de dos botones

Diseño de POEs (4)

- Diseñar un sistema de control de tiempo que permita realizar mediciones temporales en instantes relevantes de una **carrera de relevos** (atletismo, natación, patinaje sobre hielo, esquí de fondo, etc.). En concreto, se pretende realizar las siguientes mediciones de tiempo para un equipo:
  1. El tiempo total invertido por éste hasta completar la carrera
  2. El tiempo invertido por cada miembro del equipo de relevos, desde el momento en que se le entrega el testigo (o, si es el primer competidor, desde que da comienzo la carrera) hasta que hace entrega de éste al siguiente participante (o finalice la carrera si es el último competidor)

Diseño de POE (5)

- El botón *start/stop* inicia la cuenta de tiempo transcurrido, las sucesivas pulsaciones del mismo pararán o continuarán dicha cuenta
- Si se pulsa el botón *lap* con el cronómetro arrancado se congela el tiempo que se muestra en el *display*, una segunda pulsación descongela el tiempo que se muestra en el *display* mostrándose el tiempo realmente transcurrido
- Si se pulsa el botón de *lap* con el cronómetro parado se reinicia el tiempo a 0, siendo ésta la única forma de volver a la situación de partida.

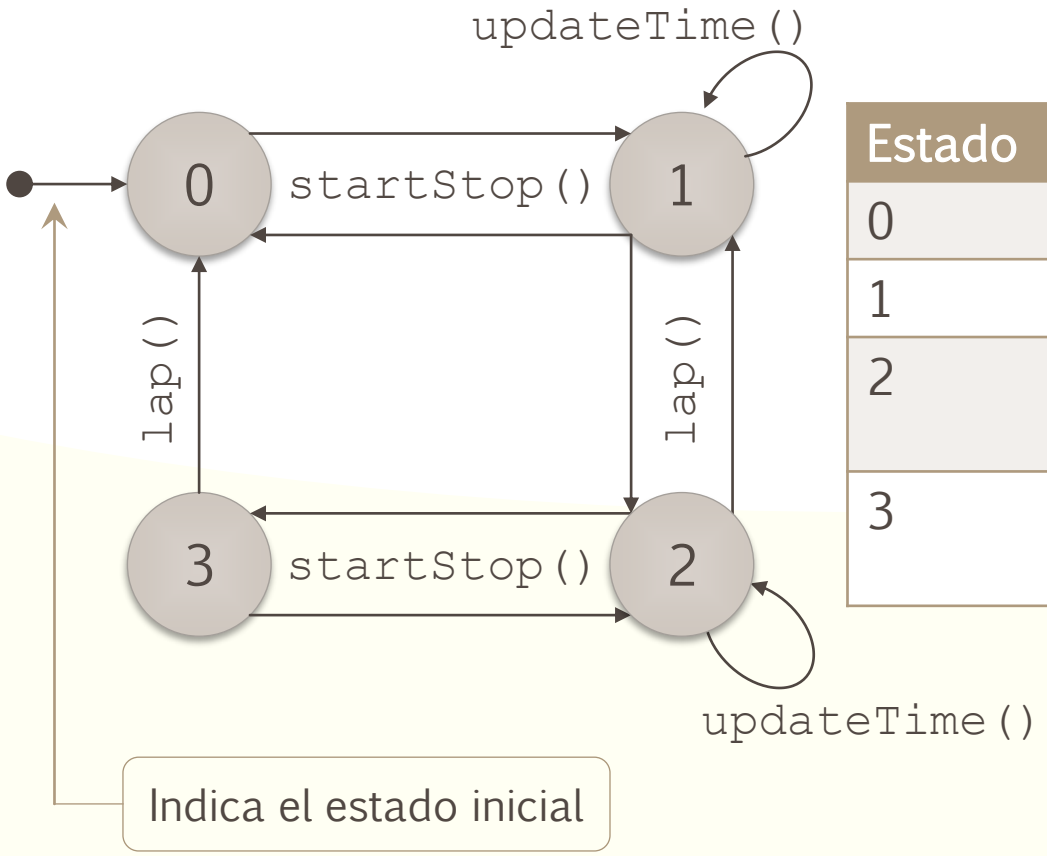
Diseño de POE (6)

- Eventos del sistema

Evento	Descripción
$startStop()$	Arranca o para el cronómetro
$lap()$	Se ha pulsado el botón de vuelta
$updateTime()$	Evento interno generado por un temporizador cada décima de segundo

Diseño de POE (7)

- Grafo de estado



Estado	Descripción
0	Cronómetro parado
1	Cronómetro arrancado
2	Cronómetro arrancado con el display congelado
3	Cronómetro parado con el display congelado

Diseño de POE (8)

- Grafo de estado en formato tabular

Eventos	0	1	2	3
$startStop()$	1	0	3	2
$lap()$	ANP	2	1	0
$updateTime()$	-	1	2	-

  - Eventos que lanzan la excepción ANP (*A*cción *N*o *P*ermitida): $lap()$
  - El estado de control está dado por la variable que toma los valores 0, 1, 2 y 3

Diseño de POE (9)

- Estado de datos
  - Un entero t que contabiliza las décimas de segundo transcurridas cuando el cronómetro está arrancado. Valor inicial: 0

- Tabla de acciones

Estados	$starStop()$	$lap()$	$updateTime()$
0			
1			$t = t + 1$ $display(t)$
2		$display(t)$	$t = t + 1$
3		$t = 0$ $display(t)$	

Infraestructura de Eventos (1)

- Infraestructura de eventos
  - Infraestructura de servicios de eventos que incluyen los lenguajes y las bibliotecas que admiten la POE para facilitar el desarrollo y la ejecución de programas basados en eventos
  - **Bucle de eventos**

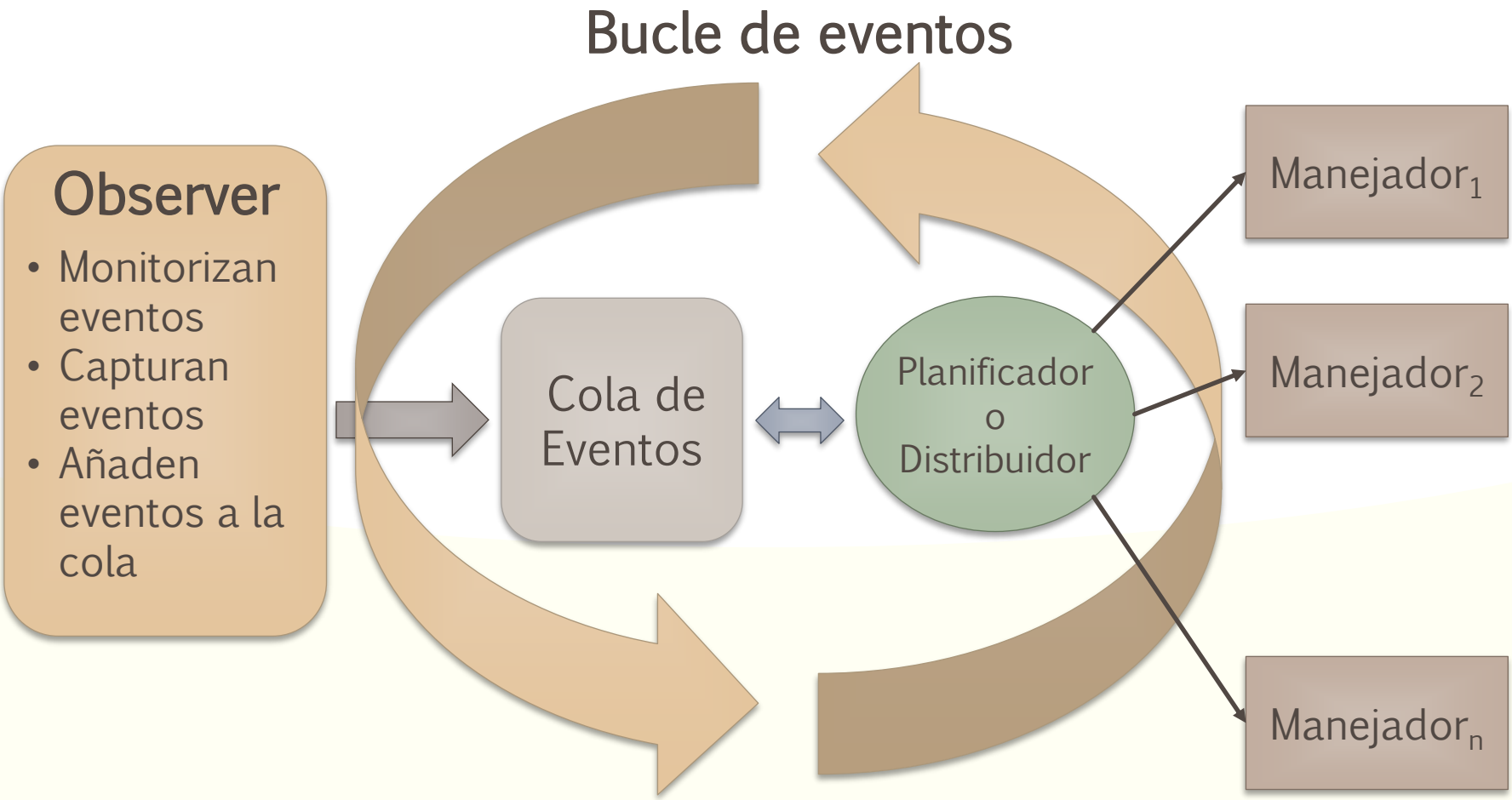
Infraestructura de Eventos (2)

- Requisitos de la infraestructura
  - Requisitos del manejo de eventos
    - En algún momento después del evento, el sistema logra un estado consistente con los requisitos de comportamiento definido para el evento.
  - Requisitos de tiempo
    - Suele ser habitual que se establezcan ciertos plazos para completar el procesamiento de un evento.
      - Este es un requisito primordial en Sistemas de Tiempo Real.

Infraestructura de Eventos (3)

- Manejo concurrente de eventos
  - Los eventos se procesan en lapsos de tiempo superpuestos; es decir, suceden concurrentemente.
    - En Sistemas de Tiempo Real es común encontrar múltiples procesadores dedicados que manejan eventos
    - Las computadoras de escritorio modernas tienen múltiples núcleos y el software de forma automática sólo usa uno.
      - Los lenguajes modernos, como Java, utilizan una abstracción llamada **hilo** (*thread*) para implementar concurrencia dentro de un programa.

Infraestructura de Eventos (4)



Infraestructura de Eventos (5)

- Registro de eventos
  - Un controlador o manejador de eventos registra su interés en un evento particular con la fuente del evento
  - La fuente del evento realiza un seguimiento del controlador hasta que se elimina el registro o la aplicación finaliza
    - *Unidifusión*. Un controlador por evento
    - *Multidifusión*. Múltiples controladores por evento

Infraestructura de Eventos (6)

- El distribuidor de eventos
  - Es el responsable de invocar a los controladores de eventos cuando se dispara un evento
  - Existen dos enfoques de envío de eventos: *push* y *pull*
    - En el envío directo, la fuente del evento es responsable de activar el distribuidor cuando ocurre un evento. La fuente del evento empuja (*push*) éste al distribuidor.
      - Cabe la posibilidad de que la fuente del evento llame directamente al manejador (la fuente también sería el distribuidor)

Infraestructura de Eventos (7)

- En la distribución de extracción, el distribuidor consulta periódicamente o sondea (*pull*) las fuentes para eventos (o la cola de eventos si existe) en búsqueda de eventos. Cuando encuentra uno, llama a los controladores relevantes

- La cola de eventos
  - En el envío directo, el distribuidor de eventos puede activar un controlador de eventos llamándolo directamente. Si el manejador no regresa lo suficientemente rápido, un segundo evento puede llegar al distribuidor mientras el primer evento todavía se está manejando.

Infraestructura de Eventos (8)

- Esto dará lugar a resultados impredecibles, bien porque se llame de nuevo al controlador de eventos, bien porque se descarte el segundo evento.
- Para solucionar este problema en la mayoría de los sistemas existe una *cola de eventos*

- En la infraestructura, el tamaño de la cola de eventos y el tiempo de ejecución de los manejadores son críticos

Concurrencia. Hilos (1)

- Concurrencia. Hilos
  - Como ya se indicó uno de los requisitos de una *Infraestructura de Eventos* es la concurrencia
    - Los eventos se procesan en lapsos de tiempo superpuestos
  - Un hilo es un mecanismo que permite implementar concurrencia en un programa
    - Es la unidad de cómputo más pequeña que el sistema operativo o la máquina virtual puede programar para ejecutar.

Concurrencia. Hilos (2)

- ¿Por qué utilizar hilos?
  - Los programas multihilo tienen varias ventajas sobre los programas que sólo disponen del hilo principal
    - Los programas con un único hilo tienen un comportamiento secuencial y, por ejemplo, el usuario siempre tendría que esperar a que se complete cierto procesamiento, lo que es un inconveniente con GUIs u otros programas interactivos.
      - Java, por ejemplo, usa un hilo separado, llamado el *hilo de envío de eventos* para manejar eventos de GUI.

Concurrencia. Hilos (3)

- Mejorar la eficiencia del programa
  - Con programación multihilo se pueden aprovechar los recursos (*cores*) de las CPUs.
  - Adicionalmente se accede a recursos compartidos, como, por ejemplo, bases de datos. Abrir y cerrar bases de datos puede requerir mucho tiempo, por lo que compartir un identificador de base de datos entre los hilos es un uso eficiente de los recursos.

Concurrencia. Hilos en Java (1)

- Hilos en Java
  - Son objetos que ejecutan, en concurrencia con el resto del programa, el código predefinido en el método `run()`.
  - Se pueden crear de dos formas:
    - Heredando de la clase `Thread`
    - **Implementando la clase `Runnable`**
  - Son procesos ligeros
    - Comparten memoria

Concurrencia. Hilos en Java (2)

- Comienzo de ejecución
  - `start()`. Arranca la ejecución concurrente del método `run`.
- Terminación
  - Retorno del método `run`
  - Por lanzarse una excepción no capturada

Concurrencia. Hilos en Java (3)

- Herencia de la clase `Thread`
  - Permite instanciar objetos que se ejecutan en un hilo separado
  - La clase en la que se instancian los objetos derivan de `Thread` y no pueden heredar de otra clase
- Implementación de la interfaz **`Runnable`**
  - Cualquier clase que implemente esta interfaz puede ejecutarse como un hilo separado
    - Permite dotar a las clases que derivan de otras de capacidad de ejecución concurrente

Concurrencia. Clase Thread (1)

Atributos	
public static final int	`MIN_PRIORITY` La prioridad mínima que un hilo puede tener.
public static final int	`NORM_PRIORITY` La prioridad por defecto que se le asigna a un hilo.
public static final int	`MAX_PRIORITY` La prioridad máxima que un hilo puede tener.

Concurrencia. Clase Thread (2)

Constructores	
public	`Thread()` Crea un nuevo objeto Thread. Se le asigna el nombre Thread-*num*, donde *num* es un entero asignado consecutivamente
public	`Thread(String name)` Crea un nuevo objeto Thread, asignándole el nombre name.
public	`Thread(Runnable target)` Crea un nuevo objeto Thread, `target` es el objeto que contiene el método `run()` que será invocado al lanzar el hilo con `start()`.
public	`Thread(Runnable target, String name)` Como el previo, pero asignando un nombre.

Concurrencia. Clase Thread (3)

Métodos	
public static Thread	`currentThread()` Retorna una referencia al hilo que se está ejecutando actualmente.
public String	`getName()` Retorna el nombre del hilo.
public int	`getPriority()` Retorna la prioridad del hilo.
public final boolean	`isAlive()` Chequea si el hilo está vivo. Un hilo está vivo si ha sido lanzado con `start()` y no ha muerto todavía.
public final void	`join() throws InterruptedException` Espera a que este hilo muera.

Concurrencia. Clase Thread (4)

Métodos	

public final void	`join (long millis)`
	`throws InterruptedException`
	Espera como mucho `millis` milisegundos para que este hilo muera.

public void	`run()` Si este hilo fue construido usando un objeto que implementaba `Runnable`, entonces se llama al método run de ese objeto. En cualquier otro caso este método no hace nada y retorna.
public final void	`setName(String name)` Cambia el nombre del hilo por `name`.
public final void	`setPriority(int newPriority)` Asigna la prioridad `newPriority` a este hilo.

Concurrencia. Clase Thread (5)

Métodos	

public static void	`sleep(long millis)`
	`throws InterruptedException`
	Hace que el hilo que se está ejecutando actualmente cese su ejecución el tiempo especificados pasando al estado dormido. El hilo no pierde la propiedad de ningún cerrojo que tuviera adquirido con `synchronized`.

public void	`start()` Hace que este hilo comience su ejecución. La MVJ llamará al método `run` de este hilo.
public static void	`yield()` Hace que el hilo que se está ejecutando actualmente pase al estado listo. Otro hilo gana el procesador.

Concurrencia. Estados de un hilo (1)

- Estados de un hilo
- **enum** `Thread.State`
  - `NEW`: creado pero no arrancado.
  - `RUNNABLE`: se está ejecutando en la JVM, aunque en un determinado momento no tenga el procesador.
  - `BLOQUED`: bloqueado esperando por un semáforo, entrar en una zona `synchronized`, o continuar tras un `wait`.

Concurrencia. Estados de un hilo (2)

- `WAITING`: esperando indefinidamente a que otro hilo realice alguna acción.
  - Se invocó `wait()` y se espera por un `notify()`
  - Se invocó `join()` y se espera por el fin de otro hilo.
- `TIMED_WAITING`: esperando a que otro hilo realice alguna acción, pero con un temporizador.
  - Se invocó `sleep`, `wait` o `join` con un tiempo.
- `TERMINATED`: terminada su ejecución.

Concurrencia. Problemas inherentes (1)

- Exclusión mutua
  - Cuando los hilos acceden a recursos compartidos y los modifican, los resultados obtenidos son impredecibles si no se garantiza el acceso exclusivo a éstos.
    - Sólo un hilo debe ejecutar una *sección crítica* para escritura, bloqueando otros lectores o escritores
    - Si el acceso a la *sección crítica* es de sólo lectura puede haber varios lectores accediendo simultáneamente, pero se ha de bloquear cualquier escritor

Concurrencia. Problemas inherentes (2)

```
public class EjemploShare implements Runnable {
    private int n = 0;
    . . .
    public void run() {
        for (int i = 0; i < 1000; i++) {
            n++;
            n--;
        }
    }
    public static void main(String[] args) {
        EjemploShare r = new EjemploShare();
        new Thread(r).start();
        new Thread(r).start();
        . . .
```

Sección Crítica

Concurrencia. Problemas inherentes (3)

- Exclusión mutua con semáforos

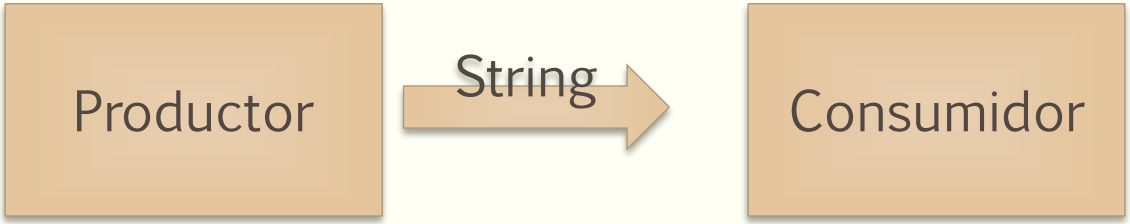
```
public class EjemploShare implements Runnable {
    private int n = 0;
    private Semaphore mutex = new Semaphore(1, true);
    . . .
    public void run() {
        for (int i = 0; i < 1000; i++) {
            mutex.adquire();  // P(mutex)
            n++;
            n--;
            mutex.release();  // V(mutex)
        }
    }
```

Requiere un try-catch

Sección Crítica

Concurrencia. Problemas inherentes (4)

- Sincronismo
  - Aunque los hilos se ejecuten concurrentemente (por tanto de forma asíncrona) se pueden dar situaciones en que uno o más hilos deban esperar a que otro (u otros) hilo finalice cierto procesamiento o se produzca un determinado evento, surge por tanto el *sincronismo*.
    - Por ejemplo, cuando este último requiera cierta información (o datos) que producen los primeros.



Concurrencia. Corrección de programas (1)

- Cuándo es correcto un programa?
  - Si cumple las especificaciones (al igual que un programa secuencial)
  - Si satisface una serie de propiedades inherentes a la programación concurrente
    - *Propiedades de seguridad*. Son aquellas que aseguran que nada malo va a pasar durante la ejecución del programa.
    - *Propiedades de viveza.* Son aquellas que aseguran que algo bueno pasará eventualmente durante la ejecución del programa.

Concurrencia. Corrección de programas (2)

- Propiedades de seguridad
  - Exclusión mutua
    - Hay que garantizar que cuando un hilo accede a un recurso compartido, otros hilos esperarán a que sea liberado.
  - Sincronización
    - Cuando un hilo debe esperar por la ocurrencia de un evento para poder seguir ejecutándose, hay que garantizar que el hilo no prosiga hasta que se haya producido el evento.

Concurrencia. Corrección de programas (3)

- Interbloqueo (*deadlock*)
  - Hay que garantizar que no se produzcan situaciones en la que todos los hilos estén esperando por un evento que nunca se producirá.
- Propiedades de viveza
  - Interbloqueo activo *(livelock)*
    - Se produce una situación de interbloqueo activo cuando un sistema ejecuta una serie de instrucciones sin hacer ningún progreso. Hay que garantizar que no ocurra este tipo de situaciones.
      - Es difícil de detectar

Concurrencia. Corrección de programas (4)

- Inanición (*starvation*)
  - Se produce una situación de este tipo cuando el sistema en subconjunto hace progresos, pero existe un grupo de hilos que nunca progresan pues no se les otorga tiempo de procesador para avanzar. Hay que garantizar una cierta equidad en el trato a los hilos, a no ser que las especificaciones del sistema digan lo contrario.
    - También es difícil de detectar

Concurrencia. Corrección de programas (5)

- Para garantizar las propiedades de seguridad
  - Semáforos. Son el medio primitivo de garantizar las propiedades de seguridad
  - Otros mecanismos de mayor nivel (más abstractos)
    - **Regiones críticas**. Facilitan el acceso exclusivo a las secciones críticas
    - **Monitores**. Tipos abstractos de datos que garantizan el acceso exclusivo a sus valores (u objetos)
    - **Señales y paso de mensajes**. Facilitan la sincronización entre procesos concurrentes

Concurrencia. Problemas inherentes (3)

- Exclusión mutua con semáforos Java

```
public class EjemploShare implements Runnable {
    private int n = 0;
    private Semaphore mutex = new Semaphore(1, true);
    . . .
    public void run() {
        for (int i = 0; i < 1000; i++) {
            mutex.acquire();  // P(mutex)
            n++;
            n--;
            mutex.release();  // V(mutex)
        }
    }
```

Requiere un try-catch

Sección Crítica

Concurrencia. Java (2)

- Permiten sincronizar hilos
  - **`wait()`**. Hace que el hilo actual espere hasta que otro hilo invoque el método `notify()` o el método `notifyAll()` para este objeto.
  - **`notify()`**. Despierta un único hilo que esté esperando en el semáforo de este objeto.
  - **`notifyAll()`**. Despierta todos los hilos que están esperando en el semáforo de este objeto.

Concurrencia. Java (1)

- Semáforos
  - Tipo Semaphore
    - Método `adquire()` (P)
    - Método `release()` (V)
- Heradado de `Object`
  - Todos los objetos tienen un semáforo asociado
    - Permite ordenar los accesos concurrentes
    - Incluye una cola de procesos bloqueados a la espera de poder acceder al objeto
  - Métodos `wait()`, `notify()` y `notifyAll()`

Concurrencia. Java (3)

- Instrucción `synchronized`
  - Facilita las **regiones críticas** y los **monitores** en Java
  - Región crítica
    - Delimita una sección crítica garantizando la exclusión mutua

      ```
      synchronized(objeto) { // sección crítica }
      ```
  - Monitor
    - Una clase (tipo de dato) en la que cada método que lo requiera está sincronizado.

      ```
      public synchronized tipo nombre(...) {...}
      ```

Concurrencia. Java (4)

- Exclusión mutua con `synchronized`

```
public class EjemploShare implements Runnable {
    private int n = 0;
    private Object mutex = new Object();
    . . .
    public void run() {
        for (int i = 0; i < 1000; i++) {
            synchronized(mutex) {
                n++;
                n--;
            }
        }
    }
```

⇨ *Sección Crítica*

Concurrencia. Java (4)

- Ejemplo. Productores-Consumidores
  - Intercambio de información por medio de un ***buffer*** (cola) de cierto tamaño que ha de estar sincronizada.
    - No se puede guardar información en el *buffer* (método `put`) cuando está lleno
    - No se puede recuperar información del *buffer* (método `get`) cuando está vacío
  - El *buffer* es un objeto compartido, por tanto, se implementará como un monitor.
  - Productor y consumidor son procesos concurrentes diferentes.

Concurrencia. Java (5)

```
// Monitor Buffer<E>
public class Buffer<E> {
    private Deque<E> data;  // cola

    private final static int BUFFER_SIZE = 10;

    public Buffer() {
        data = new ArrayDeque<E>(BUFFER_SIZE);
    }

    public synchronized E get() {
        while (data.isEmpty()) { // buffer vacío
            wait();
        }
        notify();
        return data.removeFirst();
    }
```

Requiere un try-catch

Concurrencia. Java (6)

```
    public synchronized void put(E info) {
        while (data.size() == BUFFER_SIZE) {
            // buffer lleno
            wait();
        }
        data.addLast(info);
        notify();
    }
}
```

Requiere un try-catch

Concurrencia. Java (7)

```
public class Productor<E> implements Runnable {
    private final Buffer<E> box;
    private final Supplier<? extends E> fGenerator;

    public Productor(Buffer<E> box,
                     Supplier<? extends E> fGenerator) {
        this.box = box;
        this.fGenerator = fGenerator;
    }

    public void run() {
        while (true) {
            E data = fGenerator.get();
            box.put(data);
        }
    }
}
```

Concurrencia. Java (8)

```
public class Consumidor<E> implements Runnable {
    private final Buffer<E> box;
    private final Consumer<? super E> fConsumer;

    public Consumidor(Buffer<E> box,
                      Consumer<? super E> fConsumer) {
        this.box = box;
        this.fConsumer = fConsumer;
    }

    public void run() {
        while (true) {
            E data = box.get();
            fConsumer.accept(data);
        }
    }
}
```

Concurrencia. Java (9)

```
// Programa principal
public class ProductorConsumidor<E> {
    private Buffer<E> box;
    private Thread[] productores;
    private Thread[] consumidores;

    private final static int NUMERO_PRODUCTORES = 2;
    private final static int NUMERO_CONSUMIDORES = 3;

    public ProductorConsumidor() {
        box = new Buffer<E>();
       // arrays para los hilos productores y consumidores
        productores = new Thread[NUMERO_PRODUCTORES];
        consumidores = new Thread[NUMERO_CONSUMIDORES];
    }
```

Concurrencia. Java (10)

```
public void lanzar() {
    for (int k = 0; k < NUMERO_PRODUCTORES; k++) {
        productores[k] = new Thread(new Productor<E>(box,
            new Produccion()), "Productor-" + k);
        productores[k].start();
    }

    for (int k = 0; k < NUMERO_CONSUMIDORES; k++) {
        consumidores[k] = new Thread(new Consumidor<E>(box,
            new Consumicion()), "Consumidor-" + k);
        consumidores[k].start();
    }
}

public static void main(String[] args) {
    new ProductorConsumidor<Integer>().lanzar();
}
```