

# Programación Concurrente con Java

*Diseño de Sistemas Operativos*  
Facultad de Informática

Juan Pavón Mestras  
Dep. Sistemas Informáticos y Programación  
Universidad Complutense Madrid

## Concurrencia

- En el mundo real, muchas cosas pasan a la vez
  - Con varias computadoras se pueden ejecutar varios programas a la vez
  - Con una sola computadora se puede simular la ejecución paralela de varias actividades:
    - ? múltiples flujos de ejecución (*multithreading*)  
comparten el uso de un procesador
- Java soporta la ejecución paralela de varios *threads* (hilos de ejecución)
  - Los threads en una misma máquina virtual comparten recursos
    - por ejemplo, memoria
  - Los threads en varias máquinas virtuales necesitan de mecanismos de comunicación para compartir información

## Concurrencia

---

- **Para qué:**
  - Mejorar la disponibilidad y eficiencia
  - Modelar tareas, objetos autónomos, animación
  - Paralelismo: múltiples procesadores, simultanear E/S
  - Protección: aislar actividades en hilos de ejecución
  - Ejemplos:
    - Tareas con mucha E/S: acceso a sitios web, bases de datos
    - Interfaces gráficas de usuario: gestión de eventos
    - Demonios con múltiples peticiones de servicio simultáneas
    - Simulación
- **Con cuidado:**
  - Complejidad: seguridad, viveza, composición
  - Sobrecarga: Mayor uso de recursos

## Programación concurrente OO

---

- **La concurrencia es natural en orientación a objetos**
  - Ya Simula67 soportaba concurrencia
- **Diferencias con otros modelos de concurrencia**
  - Programación OO secuencial
    - Tiene más importancia la seguridad y la viveza
    - Pero también usa y extiende patrones de diseño comunes
  - Programación orientada a eventos
    - Permite que pueda haber múltiples eventos a la vez
    - Pero usa y extiende estrategias de mensajería
  - Programación multithread de sistemas
    - Añade encapsulación y modularidad
    - Pero usa y extiende implementaciones eficientes

## Programación concurrente OO

---

- Concurrencia y reusabilidad
  - Mayor complejidad
    - Criterios de corrección más duros que con código secuencial
    - El no determinismo impide la depuración y el entendimiento del código
  - Mayor dependencia del contexto
    - Los componentes son seguros y vivos sólo en determinados contextos: necesidad de documentación
    - Puede ser difícil extender mediante herencia (anomalías)
    - Puede ser difícil la composición: conflictos entre técnicas de control de la concurrencia

## Programación concurrente OO

---

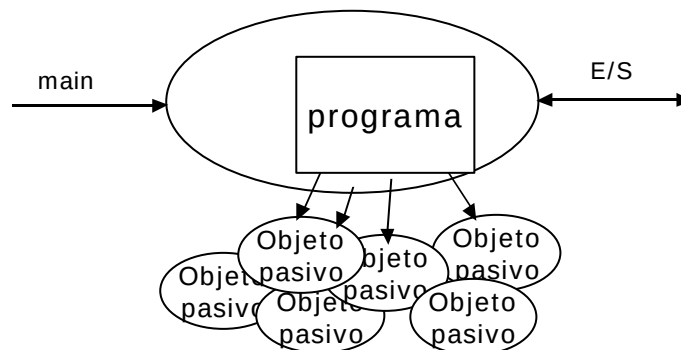
- Políticas de diseño
  - Ejemplos
    - Dependencia de estado: qué hacer cuando se recibe una petición que no se puede realizar
    - Disponibilidad de servicio: Restricciones en el acceso concurrente a los métodos
    - Restricciones de flujos: Establecer direccionabilidad y reglas de capas para los mensajes
  - Combaten la complejidad
    - Reglas de diseño de alto nivel y restricciones arquitecturales evitan decisiones caso por caso inconsistentes
  - Mantener la apertura
    - Acomodar componentes que obedecen políticas determinadas

## Modelos de objetos

- 4 operaciones básicas
  - Aceptar un mensaje
  - Actualizar el estado local
  - Enviar un mensaje
  - Crear un nuevo objeto
- Dependiendo de las reglas para estas operaciones, hay 2 categorías de modelos:
  - Objetos activos
  - Objetos pasivos

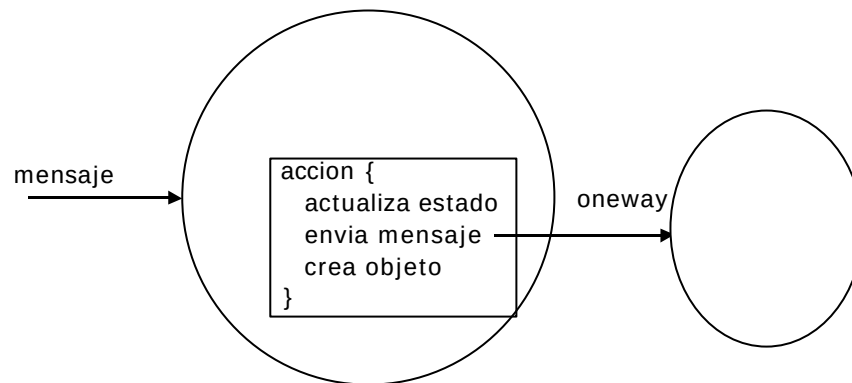
## Modelo de objetos pasivos

- En programas secuenciales, sólo un objeto Programa es activo
- Los objetos pasivos encapsulan datos del programa



## Modelo de objetos activos

- Cada objeto tiene su propio hilo de ejecución (sólo puede hacer una cosa a la vez)



## Sistemas = Objetos + Actividades

### ■ Objetos

- TADs, componentes agregados, monitores, objetos de negocio (EJBs), servlets, objetos CORBA remotos
- Se pueden agrupar de acuerdo a estructura, role, ...
- Se pueden usar para múltiples actividades
  - Lo más importante es la SEGURIDAD

### ■ Actividades

- Mensajes, cadenas de llamadas, workflows, hilos de ejecución, sesiones, escenarios, scripts, casos de uso, transacciones, flujos de datos
- Se pueden agrupar por origen, función, ...
- Comprenden múltiples objetos
  - Lo más importante es la VIVEZA

## Java

---

- Soporte para ingeniería de software
  - Empaquetado: objetos, clases, componentes, paquetes
  - Portabilidad: bytecode, unicode, transportes varios
  - Extensibilidad: Subclases, interfaces, class loader
  - Seguridad: máquina virtual, verificadores de código
  - Bibliotecas: paquetes java.\*
  - Ubicuidad: corre en casi todas partes
- Retos en nuevos aspectos de la programación
  - Concurrencia: threads, locks
  - Distribución: RMI, CORBA
  - Persistencia: Serialización, JDBC
  - Seguridad: gestores de seguridad, dominios

## Concurrencia en Java

---

- El código que ejecuta un thread se define en clases que implementan la interfaz Runnable

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

## Concurrencia en Java

---

### ■ Clase Thread

- Un hilo de ejecución en un programa
- Métodos
  - run() actividad del thread
  - start() activa run() y vuelve al llamante
  - join() espera por la terminación (timeout opcional)
  - interrupt() sale de un wait, sleep o join
  - isInterrupted()
  - yield()
  - stop(), suspend(), resume() (deprecated)
- Métodos estáticos
  - sleep(milisegundos)
  - currentTread()
- Métodos de la clase Object que controlan la suspensión de threads
  - wait(), wait(milisegundos), notify(), notifyAll()

© Juan Pavón Mestras, UCM 2001

Programación concurrente con Java

13

## Concurrencia en Java

---

### ■ Creación de Threads:

```
class MiThread extends Thread {...}  
=> new MiThread()
```

```
class MiThread implements Runnable {...}  
=> new Thread(new MiThread())
```

© Juan Pavón Mestras, UCM 2001

Programación concurrente con Java

14

## Concurrencia en Java

```
import java.lang.Math;

class EjemploThread extends Thread {
    int numero;

    EjemploThread (int n) { numero = n; }

    public void run() {
        try { while (true) {
            System.out.println (numero);
            sleep((long)(1000*Math.random()));
        } } catch (InterruptedException e) { return; } // acaba este thread
    }

    public static void main (String args[]) {
        for (int i=0; i<10; i++)
            new EjemploThread(i).start();
    }
}
```

### SALIDA

1  
4  
5  
6  
2  
9  
8  
4  
3  
0  
7  
3  
7  
2  
8  
3  
1  
9  
...

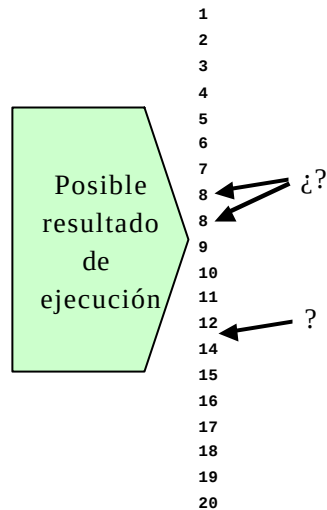
## Sincronización en Java

- Necesaria para evitar colisiones entre hilos de ejecución
    - Por ejemplo, accesos a memoria o a un recurso a la vez
  - Sincronización:
    - synchronized método (...) {...} // a nivel de objeto
    - synchronized (objeto) { ... } // a nivel de bloque de código
  - Métodos
    - wait() y wait(timeout)  
El thread se queda bloqueado hasta que algún otro le mande una señal (notify) y entonces pasa a la cola de listos para ejecutar
    - notify() y notifyAll()
- ? No es fácil la correcta programación de la concurrencia y la sincronización



## Sincronización en Java

```
public class NoSincronizada extends Thread {  
    static int n = 1;  
  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(n);  
            n++;  
        }  
    }  
  
    public static void main(String args[]) {  
        Thread thr1 = new NoSincronizada();  
        Thread thr2 = new NoSincronizada();  
  
        thr1.start();  
        thr2.start();  
    }  
}
```



## Ejercicio

- Adaptar la clase anterior para que el resultado de la ejecución de los dos threads en paralelo sea la secuencia de 1 a 20 sin repeticiones ni saltos de números.

## Monitores en Java

- Un monitor es un objeto que implementa el acceso en exclusión mutua a sus métodos
  - En Java se aplica a los métodos de la clase declarados como *synchronized*
    - Los demás métodos pueden accederse concurrentemente independientemente de si algún thread accede a ellos o a un método *synchronized*

```
// ...dentro de código synchronized, ya que el hilo debe ser el propietario del monitor
// del objeto. Liberará el monitor hasta que lo despierten con notify o notifyAll, y
// pueda retomar control del monitor
try {
    wait(); // o wait(0);
} catch (InterruptedException e) {
    System.out.println("Interrumpido durante el wait");
}
```

```
// ...también dentro de código synchronized
notify(); // o notifyAll();
```

## Monitores en Java

- Disciplinas de señalización de monitores
  - *Signal and exit*: Después de hacer *notify* el thread debe salir del monitor y el thread que recibe la señal es el siguiente en entrar en el monitor
  - *Signal and continue*: El thread que recibe la señal no tiene que ser necesariamente el siguiente en entrar en el monitor
    - ¿Puede haber intromisión: Antes que el thread despertado podría entrar un thread que estuviera bloqueado en la llamada a un método *synchronized*
- En Java:
  - Los monitores siguen la disciplina *signal and continue*
  - Tampoco se garantiza que el thread que más tiempo lleve esperando sea el que reciba la señal con un *notify()*
  - Cuando se usa *notifyAll()* para pasar todos los threads en wait a la cola de listos para ejecutar, el primer thread en entrar en el monitor no será necesariamente el que más tiempo lleve esperando

## Monitores en Java

### Ejercicio

- Realizar un programa para determinar qué disciplina siguen los monitores Java
  - Una posible manera de hacerlo es creando varios threads que llaman a un método de un objeto y dentro del método hacen un *wait()*. Otro thread será el encargado de despertarlos con *notify()* llamando a otro método del mismo objeto
  - Habrá que sacar trazas en cada método de los momentos en que un thread intenta entrar en el monitor, al hacer *wait()*, al despertarse, etc.
  - Para lograr un mayor entrelazado de los threads para este experimento sería conveniente una instrucción *sleep()* en algunos momentos, por ejemplo antes y después del *wait()* y del *notify()*. Así el planificador podrá dar entrada a otros threads.

## Propiedades de los programas concurrentes

- En programación secuencial:
  - Corrección parcial (Si el programa termina, realiza su función correctamente)
  - +
  - Terminación (el programa termina alguna vez)
  - = Corrección total
- En programación concurrente:
  - Propiedades de seguridad (Si el sistema evoluciona, lo hace correctamente)
  - Propiedades de viveza (Si algo debe ocurrir, alguna vez ocurre)
  - Equidad (Todo proceso que evoluciona lo hace recibiendo un trato equitativo de los recursos)
- ? Para verificar estas propiedades en un programa Java hay que entender cómo funciona la MVJ (y también para programar mejor...)

## Threads y Locks en la MVJ

### Conceptos para la definición del funcionamiento de la MVJ

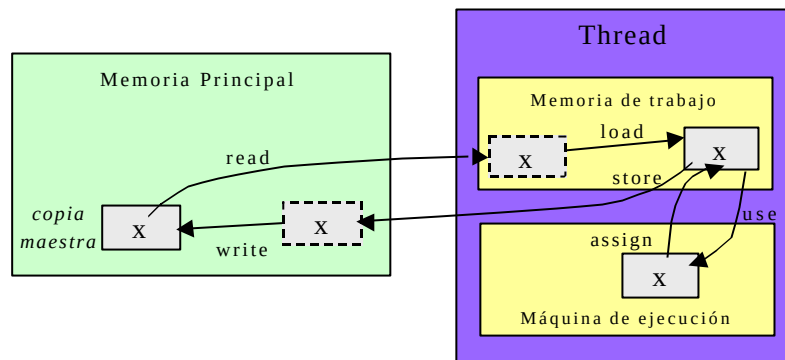
- Variables son los lugares en los que un programa puede almacenar algo
  - Variables de clases (*static*) y de objetos, y también componentes de arrays
  - Las variables se guardan en una **memoria principal** que es compartida por todos los threads
- Cada thread tiene su propia memoria de trabajo
  - El thread trabaja con copia de las variables en la memoria de trabajo
  - La memoria principal tiene una copia maestra de cada variable
- La memoria principal puede contener también cerrojos (*locks*)
  - Todo objeto Java tiene asociado un lock
  - Los threads pueden competir para adquirir un lock

## Threads y Locks en la MVJ

### Operaciones atómicas en la MVJ

- Ejecutables por threads:
  - **use** variable) (siempre que ejecute una instrucción de la MVJ que usa el valor de la variable)
  - **assign** (siempre que ejecute una instrucción de la MVJ de asignación a la variable)
  - **load** principal) (para poner en la copia de la variable el valor transmitido desde memoria principal)
  - **store** (para transmitir a memoria principal el valor de la copia de la variable)
- Ejecutables por la memoria principal
  - **read** de (para transmitir el contenido de la copia maestra de la variable a la memoria de trabajo del thread)
  - **write** memoria (para poner en la copia maestra de la variable el valor transmitido desde de trabajo del thread)
- Ejecutables en sincronización por un thread y la memoria principal
  - **lock** (para adquirir un *claim* en un cerrojo particular)
  - **unlock** (para liberar un derecho sobre un cerrojo particular)

## Acceso a variables en la MVJ



## Threads y Locks en la MVJ

- Reglas de orden de ejecución
  - Las acciones realizadas por cada thread están totalmente ordenadas
  - Las acciones realizadas por la memoria principal sobre cualquier variable están totalmente ordenadas
  - Las acciones realizadas por la memoria principal sobre cualquier cerrojo están totalmente ordenadas
  - No se permite que ninguna acción se siga a sí misma
- Relación entre las acciones de un thread y memoria principal
  - Toda acción lock y unlock se realiza a la vez por un thread y la memoria principal
  - Toda acción load sigue a una acción read
  - Toda acción write sigue a una acción store

## Threads y Locks en la MVJ

---

- Variables `long` y `double` (no declaradas como *volatile*)
  - Se consideran en la MVJ como 2 variables de 32 bits cada una
    - Son necesarias dos operaciones *load*, *store*, *read* o *write* para tratarlas
    - Para soportar procesadores de 32 bits
    - Aunque se admite que haya implementaciones de la MVJ con operaciones de 64bits atómicas (y actualmente se recomienda incluso)

## Threads y Locks en la MVJ

---

- Reglas sobre cerrojos
  - Un cerrojo sólo puede pertenecer a un thread en un momento dado
  - El cerrojo sólo quedará libre cuando el thread haga tantos *unlock* como *lock* hubiera hecho
  - Un thread no puede hacer *unlock* de un cerrojo que no poseyera
- Reglas sobre cerrojos y variables
  - Antes de hacer una operación *unlock* un thread debe copiar en memoria principal todas las variables a las que hubiera asignado
  - Después de hacer *lock* un thread debe recargar de memoria principal todas las variables que utilice

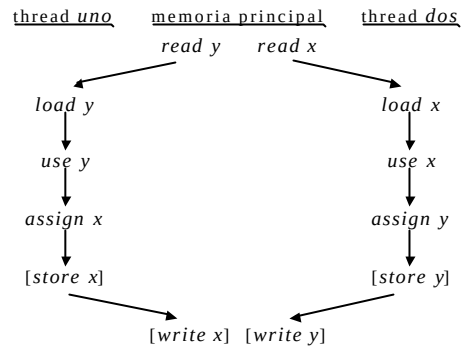
## Threads y Locks en la MVJ

```
public class Ejemplo {
```

```
    int x = 1, y = 2;
```

```
    void uno() {  
        x = y;  
    }
```

```
    void dos() {  
        y = x;  
    }  
}
```



En este ejemplo en memoria principal puede acabar ocurriendo:

- que x acabe valiendo lo que y
- o que y acabe valiendo lo que x
- o que se cambien los valores de x e y

## Threads y Locks en la MVJ

```
public class EjemploSincronizado
```

```
{
```

```
    int x = 1, y = 2;
```

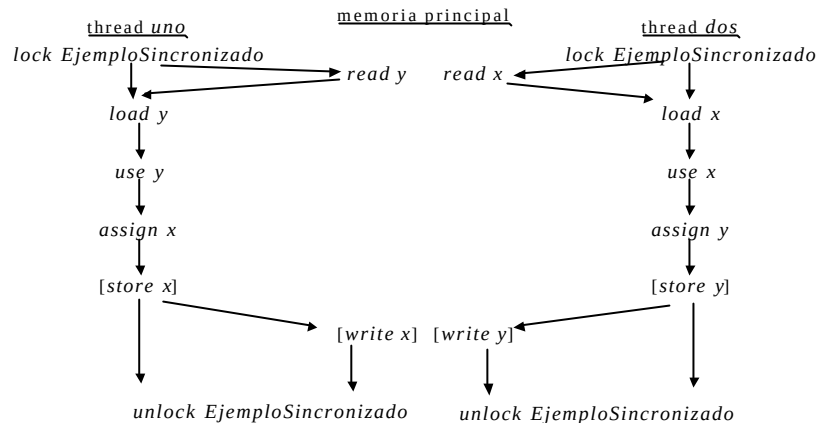
```
    synchronized void uno() {  
        x = y;  
    }
```

```
    synchronized void dos() {  
        y = x;  
    }  
}
```

Al usar la operación *unlock* se obliga a escribir en memoria principal, y por haber utilizado *lock* hay menos combinaciones, así que bien:

- o x acaba valiendo lo que y
- o y acaba valiendo lo que x
- y *no* puede haber intercambio de valores

## Threads y Locks en la MVJ



## Threads y Locks en la MVJ

### Ejercicios

- Escribir un programa que cree dos threads y pruebe la ejecución de las clases de los ejemplos anteriores
- Considerar la clase *Ejercicio2* a continuación. ¿Qué ocurre si un thread llama a *uno()* mientras otro llama a *dos()*? ¿Qué ocurre si los métodos son *synchronized*?

```
class Ejercicio2 {
    int a=1, b=2;
    void uno() { a=3; b=4; }
    void dos() { System.out.println("a = " + a + ", b = " + b); }
}
```



## Requisitos en programación concurrente OO

- Seguridad: requisitos de integridad
- Viveza: requisitos de progreso
- Eficiencia: requisitos de efectividad
- Reusabilidad: requisitos composicionales

**Políticas y protocolos**  
reglas de diseño  
a lo largo del  
sistema

**Estructuras de objetos**  
patrones de diseño  
microarquitectura

**Técnicas de codificación**  
idioms  
trucos

## Patrones de programación concurrente

- Patrones de diseño de seguridad
  - Propiedades de seguridad: *Nada malo ocurrirá*
  - Los patrones tratan básicamente de evitar que el estado del sistema se haga inconsistente
- Patrones de diseño de viveza
  - Propiedades de seguridad: *Nada malo ocurrirá*
  - Los patrones tratan básicamente de evitar que el estado del sistema se haga inconsistente

## Patrones de diseño de seguridad

---

- Objetos seguros: realizar acciones sólo cuando se está en un estado consistente
  - Conflictos lectura/escritura
  - Fallos en invariantes
- Estrategias para mantener la consistencia de estado en los objetos accedidos concurrentemente:
  - Inmutabilidad: Evitar los cambios de estado
  - Sincronización mediante cerrojos: Asegurar dinámicamente un acceso exclusivo
  - Contención: Asegurar estructuralmente un acceso exclusivo ocultando objetos internos
  - Dependencia de estado: qué hacer cuando no se puede hacer nada
  - Partición: separar los aspectos independientes de objetos y cerrojos

## Patrones de diseño de seguridad

---

- Inmutabilidad
  - Los objetos no se modifican, se crean
    - Vale para objetos que proporcionan servicios sin estado
    - Ejemplos: clases *String*, *Integer* y *Color* de Java
    - Otro ejemplo:

```
public class Punto {  
    private int x, y; // coordenadas fijas  
    public Punto(int x, int y) { this.x = x; this.y = y; }  
    // otros métodos que no cambian los valores de las  
    // coordenadas  
    public x() { return x; }  
    public y() { return y; }  
}
```

## Patrones de diseño de seguridad

---

- **Objetos sincronizados (cerrojos)**
  - Cuando el estado de los objetos puede cambiar, es necesario sincronizar el acceso a los métodos que pueden leer o modificar sus atributos
  - Básicamente puede haber dos tipos de conflictos al acceder a una variable:
    - Conflictos de lectura-escritura
    - Conflictos de escritura-escritura
  - Uso de cerrojos:
    - Adquirir el objeto cerrojo al entrar en el método, devolverlo al regresar
    - Garantiza la atomicidad de los métodos
    - Riesgo de interbloqueos

## Patrones de diseño de seguridad

---

- **Objetos sincronizados (cerrojos)**
  - En general basta con declarar todos los métodos como synchronized
    - Cada método debe ser corto de ejecución y asegurar que siempre acaba (para garantizar la viveza)
  - En algunos casos no es necesario declarar el método synchronized:
    - Métodos de acceso a valores de atributos de tipos sencillos
    - Explotar la inmutabilidad parcial
    - Arreglar la concurrencia para cada método

## Patrones de diseño de viveza

---

- Cada actividad debe progresar
  - cada método llamado tendrá que ejecutarse alguna vez
- Problemas de viveza:
  - Contención: un thread no deja el procesador
  - Reposo indefinido: un thread bloqueado nunca pasa a listo
    - Tras suspend nadie hace resume
    - Tras wait nadie hace notify
  - Interbloqueo entre threads que se bloquean uno a otro
  - Terminación prematura: un thread muere antes de lo debido (p.ej. con stop)
- Eficiencia
  - Cada método llamada debería ejecutarse lo antes posible

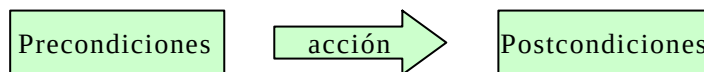
## Patrones de diseño de viveza

---

- Balance seguridad-viveza: estrategias de diseño
  - **Primero la seguridad:**
    - Asegurar que cada clase es segura (todos los métodos como synchronized) y entonces intentar mejorar la viveza para mejorar la eficiencia
      - Análisis de métodos de acceso al monitor
      - Partición de la sincronización
    - Riesgo: Puede resultar en código lento o propenso a interbloqueos
  - **Primero la viveza:**
    - Al principio no se tienen políticas de sincronización, y se añaden después con compuestos, subclases, cerrojos, etc.
    - Riesgo: puede resultar en código con errores de condiciones de carrera

## Acciones dependientes del estado

- El estado de un objeto puede tener dos tipos de condiciones de activación:
  - Internas: el objeto está en un estado apropiado para realizar una acción
  - Externas: el objeto recibe un mensaje de otro pidiéndole que realice una acción



## Acciones dependientes del estado

- Estrategia general:
  - Para cada condición que necesite esperarse, escribir un bucle guardado con un *wait()*
  - Asegurar que todo método que cambie el estado que afecte a las condiciones guardadas invoque *notifyAll()* para despertar cualquier thread que estuviera esperando un cambio de estado

```
while ( ! condicion ) {  
    try { wait(); }  
    catch (InterruptedException e) { //...  
    }  
}
```

```
// ...  
condicion = true;  
notifyAll();
```

## Acciones dependientes del estado

Ejemplo clásico: semáforo

```
public class Semaforo {  
    private int cuenta;  
  
    Semaforo(int cuentaInicial) { cuenta = cuentaInicial; }  
  
    public synchronized void P() {  
        while ( cuenta <= 0 )  
            try { wait(); }  
            catch (InterruptedException e) {}  
        cuenta--;  
    }  
  
    public synchronized void V() {  
        cuenta++;  
        notify();  
    }  
}
```

## Acciones dependientes del estado

- Políticas para tratar pre- y post-condiciones
  - Acciones ciegas: proceder en cualquier caso, sin garantías sobre el resultado
  - Inacción: ignorar la petición si no se está en estado correcto
  - Expulsión: Enviar una excepción si no se está en el estado correcto
  - Guardas: suspender hasta que se esté en estado correcto
  - Intento: proceder, ver si hubo éxito, y si no, rollback
  - Reintento: seguir intentando hasta tener éxito
  - Temporización: esperar o reintentar durante un tiempo, luego fallar
  - Planificación: iniciar primero una actividad que nos lleve a un estado correcto

## Práctica

---

- Realizar un juego que pueda ejecutarse como applet
  - El applet debe implementar la interfaz Runnable
  - Cada entidad del juego puede ser controlada por un hilo de ejecución separado:
    - Para la interfaz con de usuario
    - Jugadores
    - Supervisión del juego
    - etc.

## Bibliografía

---

- Java Virtual Machine Specification, 2nd edition
- D. Lea, Programación concurrente en Java. Principios y Patrones de diseño (segunda edición). Addison-Wesley 2000
- S. Hartley, Concurrent Programming with Java