Sincronización y herencia

- No hace falta mantener el modo sincronizado sobre-escribiendo métodos síncronos mientras se extiende una clase.
- No se puede *forzar* un método sincronizada en una interfaz.
- Sin embargo, una llamada al método de la clase súperior (con `super.`) sigue funcionando de modo síncrono.
- Los métodos estáticos también se pueden declarar `synchronized` garantizando su ejecución de manera exclusiva entre varios hilos.

Protección de miembros estáticos

En ciertos casos se tiene que proteger el acceso a miembros estáticos con un cerrojo. Para conseguir eso es posible sincronizar con un cerrojo de la clase, por ejemplo:

```
class MyClass {
  static private int nextID;
  ...
  MyClass() {
    synchronized(MyClass.class) {
      idNum=nextID++;
    }
  }
  ...
}
```

¡Ojo con el concepto!

- Declarar un bloque o un método como síncrono solo prevee que ningún otro hilo pueda ejecutar al mismo tiempo dicha región crítica (u otra sincronizada con el mismo objeto),
- sin embargo, cualquier otro código asíncrono puede ser ejecutado mientras tanto y su acceso a variables críticas puede dar como resultado fallos o efectos inesperados en el programa.

Objetos síncronos

Se obtienen objetos totalmente sincronizados siguiendo las reglas:

- todos los métodos son `synchronized`,
- no hay miembros/atributos públicos,
- todos los métodos son `final`,
- se inicializa siempre todo bien,
- el estado del objeto se mantiene siempre consistente incluyiendo los casos de excepciones.

Limitaciones para la programación concurrente

- no se puede interrumpir la espera a un cerrojo
  (una vez llegado a un `synchronized` no hay vuelta atrás)
- no se puede influir mucho en la política del cerrojo
  (distinguir entre lectores y escritores, diferentes justicias, etc.)
- no se puede confinar el uso de los cerrojos
  (en cualquier línea se puede escribir un bloque sincronizado de cualquier objeto)
- no se puede adquirir/liberar un cerrojo en diferentes flujos de control, se está obligado a una estructura de bloques

Paquete especial para la programación concurrente

- Por eso, y otros motivos, se ha introducido desde Java 5 un paquete especial para la programación concurrente.

  `java.util.concurrent`

- Hay que leer/estudiar todo su manual.
- Partes mencionaremos en clase en su momento.

Páginas del manual

Se recomienda estudiar detenidamente las páginas del manual de Java que estén relacionados con el concepto de hilo (mira también las referencias en el boletín de prácticas).

un programa concurrente

- Asumimos que tengamos un programa concurrente que quiere realizar acciones con recursos.
  (por ejemplo, los factores de la multiplicacion, o mirad las prácticas)
- Si los recursos de los diferentes procesos son diferentes no hay problema (mira por ejemplo la práctica de filtrado de matriz),
- Si dos (o más procesos) quieren manipular el mismo recurso ¿Qué hacemos?
- Vimos ya ayudas Java como `AtomicInteger`, o el `synchronized`, o los otros...
- Levantamos el nivel a algo más abstracto, revisamos los ya mencionados y luego implementamos a nivel bajo.

un programa concurrente

Tenemos básicamente tres opciones para tratar posibles escrituras concurrentes:

- se implementa exclusión mutua, es decir, solamente un proceso tiene acceso, los demás esperan;
- se implementa comportamiento idéntico, es decir, desde el algoritmo se garantiza que todos los procesos actuan igual (sobre todo: escriben lo mismo en caso que escriban concurrentemente);
- se implementa comportamiento transaccional, es decir, solo un proceso gana, lo que hacen los demás no influe en su resultado (con la opción que los demás se notifiquen en caso de fracaso) (con la opción que cualquiera o uno específico gana).

¿Qué es exclusión mutua?

- Para evitar el acceso concurrente a recursos compartidos hace falta instalar un mecanismo de control
  - que permite la entrada de un proceso si el recurso está disponible y
  - que prohibe la entrada de un proceso si el recurso está ocupado.
- Es importante entender cómo se implementan los protocolos de entrada y salida para realizar la exclusión mutua.
- Para implementar exclusión mutua se necesita algo básico a nivel hardware.
- Un método es usar un tipo de protocolo de comunicación basado en las instrucciones básicas disponibles en el hardware.
  (Eso veremos más adelante.)

estructura general basada en protocolos

Entonces el protocolo para cada uno de los participantes refleja una estructura como sigue (si protegemos código):

```
P0                          ... Pi
...                             ...
entrance protocol               entrance protocol
critical section                critical section
exit protocol                   exit protocol
...                             ...
```

Enteros con `lock`

```
import java.util.concurrent.locks.*;

// Implementation of out integer with a reentrant lock.
class Int {
  private int i;
  private ReentrantLock lock;
  Int(int i) {
    this.i=i;
    lock=new ReentrantLock();
  }
  void Add(Int I) {
    lock.lock(); // Entrance protocol.
    try {
      i+=I.i;
    } finally {
      lock.unlock(); // Exit protocol.
    }
  }
  int Get() { return i; }
}
```

conceptos

- El concepto de usar estructuras de datos a nivel alto libera al/a programador/a de los detalles de su implementación.
- Se puede asumir que las operaciones están implementadas correctamente y se puede basar el desarrollo del programa concurrente en un funcionamiento correcto de las operaciones de los tipos de datos abstractos.
- Para que se puedan utilizar con provecho hay que entender en detalle las propiedades de tales estructuras de datos.

región crítica

- Un lenguaje de programación puede realizar directamente una implementación de una región crítica.
- Así parte de la responsabilidad se traslada desde el programador al compilador.
- De alguna manera (depende del lenguaje de programación en concreto) se identifica que algún bloque de código se debe tratar como región crítica
  (así funciona Java con sus bloques sincronizados):

```
V is shared variable
region V do
  code of critical region
```

observaciones

- El compilador asegura que la variable $\mathbb{V}$ tenga un protocolo de entrada y salida adjunto que se usa para controlar el acceso exclusivo de un solo proceso a la región crítica.
- De este modo no hace falta que el programador use directamente las operaciones de los protocolos para controlar el acceso con el posible error de olvidarse de alguna parte esencial.
- Adicionalmente es posible que dentro de la región crítica se llame a otra parte del programa que a su vez contenga una región crítica. Si ésta está controlada por la misma variable $\mathbb{V}$ el proceso obtiene automáticamente también acceso a dicha región.

regiones críticas condicionales

- En muchas situaciones es conveniente controlar el acceso de varios procesos a una región crítica por una condición.
- Con las regiones críticas simples, vistas hasta ahora, no se puede realizar tal control. Hace falta otra construcción, por ejemplo:

```
V is shared variable
C is boolean expression
region V when C do
  code of critical region
```

- Mira el uso de `Condition` junto con `Lock` en Java (lo vimos en la práctica del productor/consumidor).

detalles de implementación

Las regiones críticas condicionales pueden funcionar internamente de la siguiente manera:

- Un proceso que quiere entrar en la región crítica espera hasta que tenga permiso.
- Una vez obtenido permiso comprueba el estado de la condición, si la condición lo permite, entra en la región, en caso contrario, libera el cerrojo y se pone de nuevo esperando en la cola de acceso.