Prácticas de laboratorio
Uso compartido de una piscina
(2 sesiones)

Concurrencia y Sistemas Distribuidos

Introducción

Esta práctica tiene como objetivo analizar y completar un programa concurrente en el que se aplican distintas condiciones de sincronización entre hilos. Cuando haya concluido sabrá:

- Compilar y ejecutar programas concurrentes.
- Detectar errores de sincronización en un programa concurrente.
- Analizar los requisitos de sincronización que aparecen en los programas concurrentes.
- Diseñar soluciones que cumplan con los requisitos analizados.
- Implementar dichas soluciones.

Como lenguaje de programación utilizará Java. La duración estimada de esta práctica es de dos semanas. Cada semana debe asistir a una sesión de laboratorio donde podrá comentar con el profesorado de prácticas sus progresos y resolver las dudas que le surjan. Tenga en cuenta que necesitará destinar algo de tiempo de su trabajo personal para concluir la práctica.

A lo largo de la práctica verá que hay una serie de ejercicios a realizar. Se recomienda resolverlos y anotar sus resultados para facilitar el estudio posterior del contenido de la práctica.

El problema del uso compartido de una piscina

En esta práctica se requiere modelar el funcionamiento de una piscina compartida por niños e instructores, en la que los niños aprenden a nadar bajo la supervisión de los instructores.

- Existen por tanto dos tipos de nadadores:
  - ***Kid*** *(niño),* que está aprendiendo a nadar
  - ***Instructor***, que supervisa el entrenamiento de uno o más niños
- En las instalaciones de la piscina distinguimos dos zonas:
  - Vaso de la piscina (donde se nada). - todo nadador (ya sea niño o instructor) entra en el agua y permanece un tiempo nadando hasta que sale a descansar.
  - Zona externa (terraza).- donde descansan los nadadores durante un tiempo entre baño y baño.

Todo nadador (instructor o niño) ejecuta el siguiente pseudo-código:

Nadador
repite un número de ciclos nada descansa

Cada nadador se representa mediante un hilo, y ejecutamos de forma concurrente varios hilos de tipo *Kid* y varios hilos de tipo *Instructor* (ambas cantidades son configurables). Cada vez que se nada o descansa, la duración de dicha acción es aleatoria.

El objeto compartido por los distintos hilos es la piscina (**Pool**). Para utilizar la piscina, los hilos deben invocar las operaciones correspondientes, que se comentarán más adelante.

Clase abstracta Piscina *(Pool)*

Tanto niños como instructores deben respetar las normas de uso de la piscina. Cuando un hilo intenta una acción que incumple las reglas de la piscina, dicho hilo debe esperar hasta que el estado de la piscina convierta dicha acción en legal (sincronización condicional).

Distinguimos 5 casos posibles (que modelaremos como diferentes tipos de piscinas), cada uno con normas más estrictas que el caso anterior *(ej. para Pool3 hay que cumplir también las de Pool2 y Pool1)*:

Tipo de piscina	*Reglas para K niños e I instructores*
Pool0	Baño libre (acceso libre a la piscina) *(free access)*
Pool1	Los niños no pueden nadar solos (debe haber algún instructor con ellos) *(kids cannot be alone)*
Pool2	Pueden nadar un máximo de K/I niños por instructor *(max kids/instructor)*
Pool3	No se puede superar el aforo máximo permitido de nadadores (establecido como (K+I)/2 nadadores) *(max capacity)*
Pool4	Si hay instructores esperando salir, no pueden entrar niños a nadar *(kids cannot enter if there are instructors waiting to rest)*

Por ejemplo, en piscinas tipo 1 (*Pool1*) o superior, si no hay instructores nadando y un niño desea nadar, no se le permite entrar en el agua hasta que entre a nadar un instructor.

Para la resolución de esta práctica partiremos de la aplicación que soporta piscinas de tipo 0 (Pool0). Puede descargar dicho código desde el Lessons de la práctica 1 en el sitio Poliformat de la asignatura (fichero "PPool.jar"). El objetivo de esta práctica es introducir progresivamente las modificaciones necesarias en el código para que también soporte los restantes tipos de piscinas.

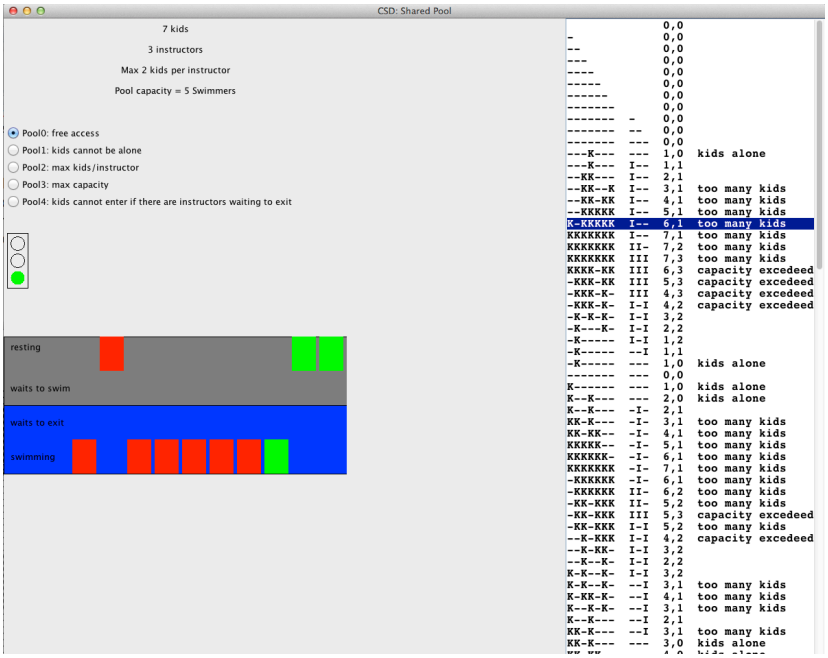
Código proporcionado

La aplicación proporcionada está completa (se puede compilar y ejecutar directamente), pero posee funcionalidad limitada (trata todos los tipos de piscina como piscinas tipo 0, o sea sin reglas). En consecuencia, no tendrá que modificar el código para realizar las primeras pruebas, pero sí en actividades posteriores.

Para trabajar con el código proporcionado, puede realizarlo desde el entorno BlueJ o desde cualquier terminal que tenga instalado un compilador Java. Por ejemplo, puede abrir en BlueJ el fichero *PPool.jar* como un nuevo proyecto (es decir, seleccione Project/Open Project y abra PPool.jar), con lo que tendrá acceso a todo el conjunto de clases que se proporcionan en la práctica. La clase *PPool* contiene el método principal *main*, que podrá lanzar a ejecución sin argumentos. Por otro lado, desde un terminal, puede ejecutar directamente la aplicación sin argumentos con la instrucción *java -jar PPool.jar*

En ambos casos, la aplicación muestra una pantalla (ver siguiente figura) en la que se puede seleccionar el tipo de piscina. Al elegir un tipo de piscina se lanza su simulación que finaliza cuando el semáforo se muestra en verde. Si al lanzar la simulación de una piscina observamos que el semáforo se queda en color rojo indefinidamente y que en la zona derecha de la pantalla se ha detenido la escritura, entonces se ha producido una situación de interbloqueo (los hilos se han quedado bloqueados entre sí).

.



La pantalla mostrada de la aplicación se estructura en las áreas que se detallan a continuación.

En la parte superior izquierda aparecen los parámetros de la simulación: 7 niños (*kids*), 3 instructores (*instructors*), máximo 2 niños por instructor, aforo (*pool capacity*) 5 nadadores.

- Por defecto se utilizan 7 niños y 3 instructores, pero al lanzar la ejecución se puede indicar como parámetro el número deseado (ej. *java -jar PPool.jar **12 4*** si deseamos 12 niños y 4 instructores).
- El número de niños debe estar en el rango [5..20], y el de instructores en el rango [2..5]. En caso contrario, el programa asume el valor por defecto.
- El máximo de niños por instructor corresponde a la operación K/I (i.e. *numNiños/numInstructores)*.
- El aforo máximo corresponde a la operación (K+I)/2 (i.e. (*numNiños + NumInstructores*)/2).

En la parte central izquierda aparecen los distintos tipos de piscina posibles. Inicialmente no hay ninguna entrada seleccionada. Al pulsar sobre alguna de las entradas, se ejecuta una nueva simulación con dicho tipo de piscina tal como se ha indicado anteriormente.

NOTA.- en el código proporcionado las piscinas tipo 1..4 se comportan igual que la 0, por lo que únicamente observaremos las diferencias al ir completando el código.

En la parte derecha aparece la secuencia de estados por los que pasa la simulación, expresados en forma de cadena de caracteres (una línea por estado). Se trata de una lista sobre la cual nos podemos desplazar y seleccionar cualquier estado. En una línea observamos distintas columnas:

- *NumNiños* caracteres (uno por niño). Los valores posibles son:
  - No ha empezado o ya ha terminado.- aparece un espacio en blanco
  - Descansa.- aparece el carácter '-'
  - Espera nadar.- aparece el carácter '*'
  - Espera salir.- aparece la letra 'k'
  - Nada.- aparece la letra 'K'
- *NumInstructores* caracteres (uno por instructor). Los valores posibles son:
  - No ha empezado o ya ha terminado.- aparece un espacio en blanco
  - Descansa.- aparece el carácter '-'
  - Espera nadar.- aparece el carácter '*'
  - Espera salir.- aparece la letra 'i'
  - Nada.- aparece la letra 'I'
- El número de niños y de instructores actualmente en el agua (incluye aquellos que nadan y aquellos que esperan salir del agua).
- Si se incumple alguna regla, aparece el correspondiente mensaje de aviso. Si dicha regla **debe** cumplirse en el tipo de piscina actual, el aviso se convierte en un mensaje de error (en **color rojo**). En dicho caso, el semáforo muestra la luz ámbar (o amarilla).
- Si todos los niños terminan, se indica el mensaje "finished". Si no quedan instructores (han finalizado todos ellos) pero hay niños que siguen activos, se indica "out of instructors".

En la parte inferior izquierda observamos la representación gráfica del estado que hayamos seleccionado en la parte derecha (es decir, la representación del estado 'actual').

- Cada nadador se representa mediante un rectángulo de color (rojo para los niños, verde para los instructores). Cada nadador siempre aparece en la misma columna y únicamente modifica su posición vertical para indicar el estado en el que se encuentra.

Cuando un nadador no ha iniciado su ejecución o ya la ha finalizado, no se visualiza su rectángulo.

- La zona con fondo gris representa la terraza, donde aparecen los nadadores que descansan (pegados a la parte superior) o intentan entrar a nadar pero deben esperar (pegados a la parte inferior).
- La zona con fondo azul representa el agua (vaso de la piscina), donde aparecen los nadadores que están nadando (pegados a la parte inferior) o están dentro del agua pero esperando salir (pegados a la parte superior).

NOTA.- en la piscina tipo 0 los nadadores no esperan nunca, por lo que aparecen siempre descansando o nadando.

Análisis del código

El código proporcionado implementa distintas clases, que podemos clasificar en:

- Clases opacas.- Necesarias para el correcto funcionamiento de la aplicación, pero sin interés para el alumno. **No deben modificarse**.
  - *State, Box, StateRenderer, Light.*

- Clases traslúcidas.- Necesitamos conocer su interfaz (para invocar operaciones desde las clases a desarrollar por el alumno), pero no necesitamos conocer su implementación. **No deben modificarse.**
  - *Pool*.- Clase abstracta con operaciones abstractas, **cuyo código debe ser implementado en las piscinas Pool0…Pool4**. Estas operaciones incluyen el método *init*, que permite la inicialización de los parámetros de la piscina, y las operaciones que invoca un nadador cuando desea un cambio de estado (ver el código de *Swimmer*, *Instructor* y *Kid*)

```
public abstract void init(int ki, int cap);
public abstract void kidSwims() throws InterruptedException;
public abstract void kidRests() throws InterruptedException;
public abstract void instructorSwims()  throws InterruptedException;
public abstract void instructorRests()  throws InterruptedException;
```

  - *Log*.- Clase con operaciones que deben invocarse cuando hay un cambio de estado de un nadador (ej. cuando se queda esperando para nadar, cuando entra en la piscina, etc.) De esta forma la aplicación podrá mostrar el estado correcto de los nadadores.
    En esta práctica deberá hacer invocaciones a los siguientes métodos de esta clase:

```
public void waitingToSwim()
public void swimming()
public void waitingToRest()
public void resting()
```

- Clases transparentes.- Interesa profundizar en su implementación (al menos en algún aspecto de la misma). Pero **no deben modificarse**.
  - *Swimmer*.- clase abstracta de la que derivan *Instructor* y *Kid*. Su método *run*() corresponde al pseudo-código indicado al principio de este texto

```
public abstract class Swimmer extends Thread{
  final int DELAY=60;
  Random rd=new Random();
  Pool pool; //piscina utilizada
  protected void delay() throws InterruptedException {
        Thread.sleep(DELAY+rd.nextInt(DELAY));} }
  public Swimmer(int id0, Pool p) { super(""+id0); pool=p;}
```

```
  public void run() {     //código que ejecuta
     try{
        pool.begin(); delay();
        for (int i=0; i<6 && !this.isInterrupted(); i++) {
           swims();  delay();
           rests();   delay();
        }
        pool.end();
    }catch (InterruptedException ex) {}
  }
   //a implementar en Instructor y en Kid
  abstract void swims() throws InterruptedException;
  abstract void rests() throws InterruptedException;
}
```

- *Instructor*

```
public Instructor(int id, Pool p) {super(id,p);}
void swims() throws InterruptedException{pool.instructorSwims(); }
void rests() throws InterruptedException {pool.instructorRests(); }
```

- *Kid*

```
public Kid(int id, Pool p) {super(id,p);}
void swims() throws InterruptedException { pool.kidSwims(); }
void rests() throws InterruptedException {pool.kidRests(); }
```

- *PPool*.- Es la **clase principal**. De esta clase interesa conocer cierto código que se realiza en el método *simulate,* en concreto:
  - la inicialización de la piscina, llamando al método *init* de la clase Pool, donde se pasa el máximo número de niños por instructor (KI) y el aforo máximo de la piscina (CAP).

```
p.init(KI,CAP);
```

  - la creación y arranque de los hilos:

```
// K=number of kids, I=number of instructors
Swimmer[] sw= new Swimmer[K+I]; //declara y crea nadadores
....
for (int i=0; i<K+I; i++)
    sw[i]= i<K? new Kid(i,p): new Instructor(i,p);
....
for (int i=0; i<K+I; i++)
    sw[i].start(); //arranca los nadadores
```

- *Pool0*.- Implementa una piscina de acceso libre (**tipo 0**). Todo tipo de piscina debe extender la clase abstracta ***Pool***. Observe cómo se utilizan en esta clase los métodos de la clase *Log* (objeto *log*).

```
public class Pool0 extends Pool {
  public void init(int ki, int cap)     {}
  public void kidSwims() { log.swimming(); }
  public void kidRests() { log.resting();}
  public void instructorSwims() {log.swimming();}
  public void instructorRests() {log.resting();}
}
```

- **Clases a modificar**.- Clases cuyo código debemos completar.
  - *Pool1, Pool2, Pool3, Pool4*. Actualmente implementan el mismo código que *Pool0*. Las actividades 1..4 indican qué debe implementar cada una de esas clases.

Actividad 0 (Pool0)

Lance varias veces la ejecución de *PPool* con piscina tipo 0 (baño libre, o sea sin reglas). Observe que la evolución de cada ejecución es distinta a la anterior (distinta duración de los tiempos para nadar/descansar, posiblemente decisiones distintas a nivel de planificación, etc.). En algunos casos el uso de la piscina incumpliría las reglas de piscinas más estrictas (tipo 1,2,3).

La siguiente tabla muestra los distintos métodos de *Pool0*, y para cada uno cuándo hay que esperar (es decir, que no se puede completar el método porque el estado de la piscina no lo permite), cómo modifica el estado de la piscina, y a quién hay que avisar una vez modificado dicho estado:

Pool0	hay que esperar si …	Modifica estado …	Avisa a …
kidSwims()	*no espera nunca*	*No hay estado*	*Nadie*
kidRests()	*no espera nunca*	*No hay estado*	*Nadie*
instructorSwims()	*no espera nunca*	*No hay estado*	*Nadie*
instructorRests()	*no espera nunca*	*No hay estado*	*Nadie*

Observe que en la piscina tipo 0 no hay un estado en la piscina que pueda cambiar e impedir o no que se complete determinada operación (siempre es posible completar la operación, y por lo tanto nunca hay que esperar). En consecuencia, no se utiliza sincronización condicional. Como ningún hilo esperará nunca, tampoco es necesario reactivar a ningún hilo en espera en ningún caso.

Preguntas Piscina Tipo 0 (baño libre):

1) Compruebe si se ha utilizado la palabra *synchronized* al implementar las operaciones de la piscina de tipo 0 (*kidSwims*, *kidRests*, *instructorSwims*, *instructorRests*). ¿Se observaría alguna diferencia de ejecución entre utilizar dicha palabra *synchronized* o no utilizarla?

No, no se ha utilizado synchronized. Esto quiere decir que los metodos actuaran sobre los objetos sin exclusión mútua. Esto crearía condiciones de carrera si los metodos modificasen variables comunes, pero no es el caso

2) ¿Se pueden producir condiciones de carrera? ¿Por qué?

No, porque no hay variables que se modifiquen aparte de la del objeto log, quien se maneja por propia cuenta (no hace falta prevenir la exclusión para este objeto)

3) ¿Cómo se ha representado el estado interno de la piscina? ¿Por qué así?

El estado interno de la piscina se ha representado con brackets vacíos, ya que la pistina de tipo 0 no tiene reglas.

4) En la piscina de tipo 0, ¿qué transiciones de estados se pueden producir? ¿Pueden los niños e instructores pasar de "resting" a "swimming" directamente, sin pasar por estados intermedios? ¿Por qué?

Sí, ya que nunca tienen que esperar

Actividad 1 (Pool1)

Ejercicio 1.1: Indique cómo representar el estado de la piscina de tipo 1 para poder cumplir las normas de uso asociadas a dicha piscina. Para ello, complete la tabla correspondiente a *Pool1*:

Pool1	hay que esperar si …	modifica estado …	Avisa a …
kidSwims()	*No hay instructores nadando*	kidNum	a nadie
kidRests()	no espera nunca	kidNum	a nadie
instructorSwims()	no espera nunca	insNum	nenes
instructorRests()	no espera nunca	insNum	nenes

Para hacer esperar a los hilos que solicitan una operación contraria a las reglas, utilizamos el siguiente esquema:

```
while (condición de esperar) {
      wait();
}
Actualiza estado
Avisa del nuevo estado a otros hilos que puedan estar esperando dicho
cambio
```

Como la instrucción wait() puede retornar una interrupción (si el hilo es interrumpido durante su espera), en la declaración de los métodos kidSwims(), kidRests(), instructorSwims(), instructorRests() se debe añadir la expresión "throws InterruptedException" cuando así fuera necesario.

Por ejemplo, el método *kidSwims* de la clase *Pool1* queda como sigue (**el alumno debe completar lo que falta**):

```
public synchronized void kidSwims() throws InterruptedException {
     while (condición de espera) {  //COMPLETAR la condición
           log.waitingToSwim();//para visualizar la posición del nadador
           wait();
      }
      ...   //Actualiza estado   (COMPLETAR)
      ...   //Si necesario, avisa del nuevo estado a otros hilos
            // con notifyAll();
      log.swimming();  //para visualizar la posición del nadador
}
```

Ejercicio 1.2: Modifique el método *kidSwims* de la clase *Pool1* para que el hilo invocante espere si no hay instructores nadando (tal y como se indica en la tabla, y siguiendo el esquema anteriormente descrito). Utilice las variables internas que considere oportunas para **contabilizar los niños e instructores** que entran/salen de la piscina. Si es necesario, modifique otros métodos de Pool1 para actualizar dichas variables.

Compile y ejecute. Observará que los niños que desean nadar cuando no hay instructores en el agua se comportan correctamente (esperan en lugar de entrar), pero sigue habiendo situaciones ilegales.

Ejercicio 1.3: Analice la traza de la ejecución y determine qué problemas o situaciones ilegales aparecen y qué situaciones sí que se han resuelto.

Ejercicio 1.4: Revise el resto de métodos de la clase *Pool1* (es decir, *kidRests*, *instructorSwims*, *instructorRests*) y modifique aquellos que sea necesario, para reflejar las entradas/salidas a la piscina de los niños e instructores de forma apropiada.

Ejercicio 1.5: Compruebe que las reglas de la piscina tipo 1 se cumplen ahora. Repita la ejecución modificando el número de niños y/o instructores. Compruebe que la ejecución continúa siendo correcta.

¡ Importante ! Compruebe que no se queda nunca bloqueado el programa en la ejecución de la piscina 1. Si así fuera, revise las condiciones de espera y de activación de los hilos.

Recuerde que el **color del semáforo** indica:

- Semáforo verde: ejecución correcta. Se cumplen todas las normas de la piscina.
- Semáforo amarillo: la ejecución ha terminado, pero alguna de las normas de la piscina no se cumple.
- Semáforo rojo: alguno de los hilos (niños o instructores) se ha quedado bloqueado indefinidamente, por lo que la ejecución no podrá finalizar.

Preguntas Piscina Tipo 1 (los niños no pueden nadar solos):

1) Compruebe si se ha utilizado la palabra *synchronized* al implementar las operaciones de la piscina de tipo 1. ¿Se observaría alguna diferencia de ejecución entre utilizar dicha palabra *synchronized* o no utilizarla?

Sí, ya que a diferencia de la piscina de tipo 0, en este tipo de piscina sí que pueden producirse condiciones de carrera, por lo que se necesita que los métodos se ejecuten en exclusión mutua.

2) ¿Se pueden producir condiciones de carrera? ¿Por qué?

Sí, ya que se trata con variables comunes a todos los objetos. Por ejemplo, sin el uso del campo synchronized, la actualización del número de instructores puede ser incorrecta

3) ¿Cómo se ha representado el estado interno de la piscina? ¿Por qué así?

se ha representado declarando un número de instructores y un número de niños

4) Para avisar del nuevo estado, se ha empleado *notifyAll*(). ¿Podríamos haber utilizado simplemente *notify*()? ¿Por qué?

No, ya que todos los hilos deben ser conscientes de los cambios, y notify solo le hace saber estos cambios al hilo sobre el que se ejecuta el método

5) ¿Al final de qué métodos de Pool1 ha empleado *notifyAll*()? ¿Lo ha utilizado al final de todos los métodos? Si es así, analice si es obligatorio emplearlo para todos los métodos, o si sería posible (o más eficiente) utilizarlo solamente en unos cuantos (en los requeridos).

no hace falta usar notifyAll en todos los métodos, solo en aquellos necesarios (los

Actividad 2 (Pool2)

Ejercicio 2.1: Indique cómo representar el estado de la piscina para poder cumplir las normas de uso indicadas para la piscina tipo 2 (recuerde que también debe cumplir con las normas de la piscina tipo 1). Para ello, rellene la tabla correspondiente a *Pool2*:

Pool2	hay que esperar si …	modifica estado …	Avisa a …
kidSwims()	no hay monitores/los suficientes	kids++	nadie
kidRests()	no hay que esperar	kids--	nadie
instructorSwims()	no hay que esperar	swim++	nenes
instructorRests()	si hay mucho niño para poco instructor	swim++	instructores

Ejercicio 2.2: Modifique el código de *Pool2* para que se cumplan las reglas de utilización de la piscina tipo 2.

Ejercicio 2.3: Verifique el funcionamiento de *Pool2* modificando el número de niños y/o instructores.

Preguntas Piscina Tipo 2 (máximo niños por instructor):

1) Para representar el estado del objeto compartido, ¿bastaría con utilizar una variable de tipo entero (ej. *nSwimKids*) y una variable de tipo booleano (ej. *instInPool*)? ¿Por qué?

Bastaría con nSwimKids, pero no con instInPool.

2) ¿A qué métodos afecta esta nueva regla? (es decir, ¿en qué métodos de Pool2 ha realizado modificaciones?)

en el 1 y el 4 método

3) Cuando un instructor entra en la piscina Pool2, ¿se debe invocar *notifyAll*()? ¿Por qué?

sí, ya que el resto de instructores deben saber si pueden descansar o no

Actividad 3 (Pool3)

Ejercicio 3.1: Indique cómo representar el estado de la piscina de tipo 3 para poder cumplir con todas sus normas de uso. Para ello, rellene la tabla correspondiente a *Pool3*:

Pool3	hay que esperar si …	modifica estado …	Avisa a …
kidSwims()	hay 5 personas ya en la piscina o no hay instructores		nadie
kidRests()	no tiene que esperar		
instructorSwims()	si hay 5 personas en la piscina		
instructorRests()	si no hay suficientes instructores en la piscina para vigilar a todos los niños		

Ejercicio 3.2: Modifique el código de Pool3 para que se cumplan las reglas de utilización de la piscina tipo 3.

Ejercicio 3.3: Verifique el funcionamiento de Pool3 modificando el número de niños y/o instructores.

Preguntas Piscina Tipo 3 (capacidad máxima):

1) Para representar el estado de la piscina, ¿se requiere añadir alguna otra variable al estado de la piscina respecto a la implementación de Pool2? ¿Cuál? ¿Por qué?

sí, una variable max

2) ¿Podemos decir que la piscina Pool3 está actuando como un "monitor"? ¿Y las piscinas anteriores?

sí

Actividad 4 (Pool4)

Ejercicio 4.1: Indique cómo representar el estado de la piscina de tipo 4 para poder cumplir con todas sus normas de uso. Para ello, rellene la tabla correspondiente a *Pool4*:

Pool4	hay que esperar si …	modifica estado …	Avisa a …
kidSwims()			
kidRests()			
instructorSwims()			
instructorRests()			

Ejercicio 4.2: Modifique el código de Pool4 para que se cumplan las reglas de utilización de la piscina tipo 4.

Ejercicio 4.3: Verifique el funcionamiento de *Pool4* modificando el número de niños y/o instructores.

Preguntas Piscina Tipo 4 (si instructores esperan, no pueden entrar niños):

1) Para representar el estado de la piscina, ¿se requiere añadir alguna otra variable al estado de la piscina respecto a la implementación de Pool3? ¿Cuál? ¿Por qué?

2) ¿Al final de qué métodos de Pool4 ha empleado *notifyAll*()? ¿Lo ha utilizado al final de todos los métodos? Si es así, analice si es obligatorio emplearlo para todos los métodos, o si sería posible (o más eficiente) utilizarlo solamente en unos cuantos (en los requeridos).