Universidad Zaragoza

1542

30322 – Programación de Redes y Servicios

Prácticas de Laboratorio

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2019-2020

ESCUELA DE INGENIERÍA Y ARQUITECTURA

Departamento de Informática e Ingeniería de Sistemas

Área de Lenguajes y Sistemas Informáticos

v 2.1

Programación de Redes y Servicios
Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación
Universidad de Zaragoza, curso 2019/2020

Práctica 1: **Introducción al modelo de concurrencia de Java**

Objetivos y requisitos

Objetivos

- Familiarización con el entorno de programación.
- Familiarización con la creación de un programa concurrente en Java: *threads*.
- Familiarización con el concepto de entrelazado (*interleaving*).
- Analizar el *Problema de la Sección Crítica*: propiedades para asegurar la corrección de programas concurrentes.
- Estudiar y analizar los *Algoritmos de espera activa* para el Problema de la Sección Crítica.

Requisitos

- Compilador y máquina virtual de Java (JVM) Java Platform Standard Edition 7[1]
- `Eclipse`, como entorno de desarrollo[2].

1. Entorno de desarrollo: `Eclipse`

En esta asignatura, utilizaremos Eclipse como entorno de desarrollo a lo largo de todas las prácticas. Eclipse ya se utiliza en la asignatura de *Fundamentos de Informática* del primer cuatrimestre de la titulación. Los elementos básicos de Eclipse se describen en una guía que ya te proporcionaron en primero y que podrás encontrar en Moodle.

[1] Disponible para descarga en `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.
[2] Disponible para descarga en `http://www.eclipse.org/downloads/`

En esencia, tenéis que aseguraros de que sabéis cómo crear un proyecto en Eclipse y cómo crear clases dentro de un proyecto. En caso de querer volver a refrescar estos conocimientos, puedes consultar algún tutorial de Internet como `https://www.vogella.com/tutorials/Eclipse/article.html`.

2. Introducción a Programación Orientada a Objetos

La programación orientada a objetos (POO) es un paradigma de programación que surge como necesidad para intentar dominar la complejidad innata que posee el software. Hasta ahora, te has enfrentado a esta complejidad mediante *programación estructurada* (también llamada *programación proceduramental*; lo que aprendiste en clase de *Fundamentos de Informática*), un tipo de paradigma de programación consistente en descomponer el problema a solucionar en varios subproblemas, de modo que cada subproblema puede resolverse con acciones muy simples y fáciles de implementar en un lenguaje de programación estructurada. Por ejemplo, para conocer si un número entero es par o impar, podemos proceder de la siguiente manera: primero, dividimos el número entre dos guardándonos su resto. Por las propiedades de los números enteros, este resto será 0 ó 1; después, comprobamos cuál es el valor del resto. En caso de que sea el valor 0, sabemos que el número entero era par. En caso contrario, el número era impar.

En los lenguajes de programación estructurada, cada uno de estos subproblemas se implementaban mediante procedimientos o funciones, que son el conjunto de instrucciones que operan sobre unos argumentos y producen un resultado. Así, un programa no es más que una sucesión razonada de llamadas a funciones o procedimientos para resolver el problema particular.

Hasta ahora, has estado usando el lenguaje de programación Java como si fuera un lenguaje de programación estructurada. Sin embargo, Java es un lenguaje de programación orientada a objetos. La programación orientada a objetos es otra forma de descomponer problemas. Este nuevo método de descomposición es la *descomposición en objetos*. En el caso de los lenguajes orientados a objetos, como es el caso de C++ y Java, el elemento básico no va ser la función o procedimiento, sino un ente denominado *objeto*. Un objeto es la representación en un programa de un concepto. El objeto, además, contiene toda la información necesaria para abstraerlo: datos que describen sus atributos y operaciones que pueden realizarse sobre los mismos.

Un objeto no es más que un conjunto de variables (definidos mediante sus *atributos*) y *métodos* (que cambian el estado del objeto; es decir, modifican sus atributos) relacionados entre sí. Los objetos en programación se usan para modelar objetos o entidades del mundo real. Un objeto es, por tanto, la representación en un programa de un concepto, y contiene toda la información necesaria para abstraerlo: datos que describen sus atributos y operaciones que pueden realizarse sobre los mismos.

Normalmente, en el mundo real existen varios objetos de un mismo tipo. En terminología de POO, diremos que *un objeto es una instancia de una clase*. Así, una clase define el concepto abstracto, mientras que un objeto será una particular instancia de dicho concepto. En otras palabras, una clase es una plantilla que define las variables y

los métodos que son comunes para todos los objetos de un cierto tipo.

Supón por ejemplo que queremos realizar un programa software para gestionar un parking que cuenta con una cámara de reconocimiento de vehículos a la entrada y salida para controlar los accesos. Esta cámara es capaz de leer la matrícula, la marca y color del vehículo, así como el número de ocupantes que están en el vehículo. En este ejemplo, el concepto de vehículo sería una clase, mientras que la matrícula (tipo cadena), marca (tipo cadena), color del vehículo (tipo cadena) y el número de ocupantes (tipo entero) serían los atributos de la clase. En Java, la clase `Vehiculo` (los nombres de las clases los escribiremos siempre con la primera letra en mayúsculas) que estamos hablando puede definirse como sigue:

```
// Fichero Vehiculo.java
public class Vehiculo{
    // Atributos de la clase
    String  matricula;
    String  marca;
    String  color;
    int     numOcupantes;
}
```

Cada uno de los vehículos que reconoce la cámara serían entonces *objetos* (instancias) de la clase `Vehiculo`. Por ejemplo, si entrara un vehículo Skoda con matrícula 0101 DFG de color gris y dos ocupantes, el programa mantendría un objeto de la clase `Vehiculo` cuyos valores de `matricula`, `marca`, `color` y `numOcupantes` serían `"0101 DFG"`, `"Skoda"`, `"gris"` y 2, respectivamente. Del mismo modo, si fuera un BMW con matrícula 3221 AX de color rojo y un único ocupante, el objeto correspondiente tendría como valores de sus atributos `"3221 AX"`, `"BMW"`, `"rojo"` y 1.

La clase `Vehiculo` definida anteriormente no tiene ningún método asociado. Vamos a añadirle un método `imprimeValores()` para que se muestren por pantalla los valores de cada uno de los atributos de la clase:

```
// Fichero Vehiculo.java
public class Vehiculo{
    // Atributos de la clase
    String  matricula;
    String  marca;
    String  color;
    int     numOcupantes;

    // Método para imprimir los valores de los atributos
    void imprimeValores(){
        System.out.println("Matrícula: " + this.matricula);
        System.out.println("Marca: " + this.marca);
        System.out.println("Color: " + this.color);
        System.out.println("Núm. de ocupantes detectados: "
                           + this.numOcupantes);
    }
}
```

Modificador	Clase	Paquete	Cualquier otro
(por defecto)	Sí	Sí	No
`public`	Sí	Sí	Sí
`protected`	Sí	Sí	No
`private`	Sí	No	No

Tabla 1: Visibilidad de métodos y atributos, según modificador.

Fíjate que cada uno de los atributos de la clase se ha antecedido con la palabra clave `this`. Esta palabra clave hace referencia al objeto actual. Así, cuando se invoque (se llame) al método `imprimeValores` de un objeto, éste accederá a los valores de sus atributos. En este ejemplo, esta palabra clave es redundante (es decir, se puede eliminar) y su eliminación no causa ningún problema a la ejecución.

Los atributos y los métodos de una clase, al igual que las propias clases, pueden tener modificaciones de su visibilidad. Estos modificadores se antecede a la definición de tipos (en los atributos) o a la definición de los tipos que se devuelven (en los métodos). Por defecto, los atributos y los métodos de una clase son visibles dentro del mismo módulo (o paquete, en terminología de Java). Esta visibilidad se puede cambiar mediante los modificadores que se muestran en la Tabla 1.

Para ver el efecto de estos modificadores sobre los atributos y métodos, vamos a implementar una clase nueva llamada `Gestor`, donde incluiremos el método `main` (recuerda que este método es el que se ejecutará cuando se lance el programa en ejecución mediante la máquina virtual de Java). En este método añadiremos el código necesario para crear los dos objetos de la clase `Vehiculo` descritas anteriormente:

```
1  public class Gestor{ // Fichero Gestor.java
2
3      public static void main(String[] args){
4          // Creación de dos objetos de tipo Vehiculo
5          Vehiculo v1 = new Vehiculo();
6          Vehiculo v2 = new Vehiculo();
7
8          // Damos valor a cada uno de los atributos (ejemplo inicial)
9          // para el vehículo 1
10         v1.matricula    = "0101 DFG";
11         v1.marca        = "Skoda";
12         v1.color        = "gris";
13         v1.numOcupantes = 2;
14         // y para el vehículo 2
15         v2.matricula    = "3221 AX";
16         v2.marca        = "BMW";
17         v2.color        = "rojo";
18         v2.numOcupantes = 1;
19
20     }
21 }
```

Si te creas los ficheros `Vehiculo.java` y `Gestor.java` en Eclipse y los ejecutas, verás que no ocurre nada, dado que el método principal actual únicamente se encarga de crear los objetos e inicializar sus atributos conforme al ejemplo comentado anteriormente.

Las instrucciones de las líneas 6 y 7 están **creando instancias** (es decir, objetos) de la clase `Vehiculo`. En Java, para crear objetos se hace uso de la palabra clave `new` seguido del nombre de la clase y con `()`. En realidad, estas líneas están llamando al *constructor* de la clase Vehículo. Un constructor es un método perteneciente a la clase que posee unas ciertas características especiales:

- Se llama igual que la clase.
- No devuelve nada, ni siquiera void.
- Pueden existir varios, pero aceptado diferentes tipos de parámetros.
- De entre los que existan, tan sólo uno se ejecutará al crear un objeto de la clase.

Un constructor es útil para definir, por ejemplo, inicializaciones de los valores de los atributos de un objeto. Por defecto, las clases en Java siempre incorporan de manera implícita un constructor que no recibe parámetros y que no hace nada. En la clase `Vehiculo`, este constructor sería:

```
public Vehiculo()
{
}
```

Java permite redefinir también estos constructores implícitos. Modifica el código de la clase `Vehiculo` añadiendo el siguiente código:

```
public Vehiculo()
{
    System.out.println("En el constructor vacío de la clase Vehiculo.");
}
```

Ejecuta de nuevo la clase `Gestor`. En este caso, la ejecución será diferente de la anterior: verás que aparecen en la consola dos líneas con el texto *"En el constructor vacío de la clase Vehiculo."*, correspondientes a la ejecución del constructor de la clase `Vehiculo` de las líneas 6 y 7 del método `main` anterior.

Si modificas la visibilidad de los atributos de `Vehiculo` y la cambias a `private`, verás que no puedes ejecutar el código del método `main`, dado que estás accediendo a atributos privados de un objeto, es decir, estos atributos sólo tienen visibilidad dentro del propio objeto.

Ahora, vamos a modificar el método `main` anterior para añadirle algo más de lógica. Después del código de la línea 18, añade lo siguiente:

```
    // Llamamos al método imprimeValores de cada objeto Vehiculo
    v1.imprimeValores();
    v2.imprimeValores();
```

El código añadido está llamando al método `imprimeValores()` de cada uno de los objetos `v1` y `v2`. Recuerda que este método lo hemos implementado anteriormente en la clase `Vehiculo` para que cada objeto de la clase pueda mostrar por pantalla los valores de cada uno de sus atributos.

Ejecuta de nuevo la clase `Gestor` y observa el resultado. Por pantalla, verás algo similar a:

```
En el constructor vacío de la clase Vehiculo.
En el constructor vacío de la clase Vehiculo.
Matrícula: 0101 DFG
Marca: Skoda
Color: gris
Núm. de ocupantes detectados: 2
Matrícula: 3221 AX
Marca: BMW
Color: rojo
Núm. de ocupantes detectados: 1
```

Como ves, aparecen las dos líneas relativas al constructor vacío que hemos implementado antes y la secuencia de impresiones de los atributos y sus valores, para cada uno de los vehículos.

Vamos ahora a definir un constructor para inicializar los atributos de la clase `Vehiculo`. Para ello, añade el siguiente código a la clase `Vehiculo`:

```
public Vehiculo(String matricula, String marca,
                String color, int numOcupantes)
{
    this.matricula      = matricula;
    this.marca          = marca;
    this.color          = color;
    this.numOcupantes   = numOcupantes;
}
```

Fíjate que en este nuevo constructor se hace uso de la palabra clave `this`. A diferencia del caso anterior, aquí esta palabra clave no es opcional, dado que ayuda al compilador a entender que se está refiriendo al atributo del objeto y no al parámetro, que tiene el mismo nombre que el atributo. Si no estuviera `this`, las líneas de código serían redundantes porque el compilador entendería que se refieren a los parámetros (no tiene sentido decir `matricula = matricula`, se está asignando a `matricula` el mismo valor que tiene).

Modifica el código del método `main` de la clase `Gestor` para que haga uso del nuevo constructor:

```
1  public class Gestor{ // Fichero Gestor.java
2
3      public static void main(String[] args){
4          // Creación de dos objetos de tipo Vehiculo,
5          //  con inicialización de atributos
6          Vehiculo v1 = new Vehiculo("0101 DFG", "Skoda", "gris", 2);
7          Vehiculo v2 = new Vehiculo("3221 AX", "BMW", "rojo", 1);
8
9          // Llamamos al método imprimeValores de cada objeto Vehiculo
10         v1.imprimeValores();
11         v2.imprimeValores();
12     }
13 }
```

Si ahora ejecutas de nuevo la clase `Gestor`, verás que las líneas de texto informativas del constructor vacío han desaparecido, dado que el nuevo código hace uso del otro constructor añadido:

```
Matrícula: 0101 DFG
Marca: Skoda
Color: gris
Núm. de ocupantes detectados: 2
Matrícula: 3221 AX
Marca: BMW
Color: rojo
Núm. de ocupantes detectados: 1
```

Dado que un vehículo siempre tendrá por lo menos un ocupante, podríamos incluso definir otro constructor que inicializara todos los atributos según los parámetros que reciba y que coloque el valor de 1 al atributo de `numOcupantes`. El código de este constructor sería similar a:

```
public Vehiculo(String matricula, String marca,
                String color)
{
    this.matricula      = matricula;
    this.marca          = marca;
    this.color          = color;
    this.numOcupantes   = 1;
}
```

Tras esta adición en la clase `Vehiculo`, se podría volver a escribir el método `main` de la clase `Gestor` como sigue:

```
1  public class Gestor{ // Fichero Gestor.java
2
3      public static void main(String[] args){
4          // Creación de dos objetos de tipo Vehiculo,
5          //  con inicialización de atributos
6          Vehiculo v1 = new Vehiculo("0101 DFG", "Skoda", "gris", 2);
7          // Llamamos al otro constructor
8          Vehiculo v2 = new Vehiculo("3221 AX", "BMW", "rojo");
9
10         // Llamamos al método imprimeValores de cada objeto Vehiculo
11         v1.imprimeValores();
12         v2.imprimeValores();
13     }
14 }
```

Para acabar con esta introducción a POO, vamos a hablar de los métodos y atributos estáticos. Considera que queremos llevar un contador de cuántos objetos de la clase `Vehiculo` se crean sin inicializar. Una forma de hacerlo sería declarando una variable entera `contadorVacios` dentro la clase `Vehiculo`, y hacer que cada vez que se cree un nuevo vehículo mediante el constructor vacío se incremente el valor de esta variable. Pero, si en realidad un objeto es una instancia particular de una clase, ¿cómo podemos mantener esa variable y que se "mantenga" entre todos los objetos? Para esto sirve la palabra clave `static`. Este modificador se añade después de la visibilidad de un atributo o un método y antes del tipo de dato para indicar que se trata de un atributo o un método estático. A diferencia de los atributos o métodos no estáticos, los atributos (métodos) estáticos son atributos (métodos) de clase, y por tanto su forma de usarlos (de invocarlos) es ligeramente diferente.

Vamos a empezar añadiendo la variable contador a la clase `Vehiculo`. Añade en la declaración de atributos lo siguiente:

```
private static int contadorVacios = 0;
```

Observa que la variable `contadorVacios` se ha inicializado al valor de cero y que además su visibilidad es privada. Vamos ahora a añadir un par de métodos, uno de ellos será para incrementar el valor de esta variable (que lo haremos también privado, así sólo se puede acceder a él desde la propia clase) y el otro público para consultar su valor:

```
private static void incrementaContadorVacios(){
    contadorVacios = contadorVacios + 1;
}

public static int getContadorVacios(){
    return contadorVacios;
}
```

Para acabar, vamos a modificar los constructores de la clase `Vehiculo` para que el constructor vacío llame al método `incrementaContadorVacios()`:

```
    public Vehiculo()
    {
        System.out.println("En el constructor vacío de la clase Vehiculo.");
        Vehiculo.incrementaContadorVacios();
    }
```

Observa que para la llamada al método estático `incrementaContadorVacios()` la sintaxis es diferente: cuando se llama a un método estático (o se accede a una variable estática desde fuera de la propia clase donde se define), es necesario anteponer al nombre del método (o la variable) el nombre de la clase. Estos métodos y atributos estáticos reciben el nombre de *métodos y atributos de clase*.

Vamos a acabar de modificar el programa principal del Código 3 para invocar a uno de estos nuevos métodos estáticos que hemos añadido. Añade las siguientes instrucciones al final del método `main`:

```
1       System.out.println("Coches creados sin información: "
2                                  + Vehiculo.getContadorVacios());
3       // Creamos un coche sin información
4       Vehiculo v3 = new Vehiculo();
5       // y repetimos la salida
6       System.out.println("Coches creados sin información: "
7                                  + Vehiculo.getContadorVacios());
```

Observa que estás invocando al método `getContadorVacios()` a través de la clase, no a través de alguna instancia de la clase. Ahora, si ejecuta el programa principal, tu salida por consola debería de ser similar a:

```
Matrícula: 0101 DFG
Marca: Skoda
Color: gris
Núm. de ocupantes detectados: 2
Matrícula: 3221 AX
Marca: BMW
Color: rojo
Núm. de ocupantes detectados: 1
Coches creados sin información: 0
En el constructor vacío de la clase Vehiculo.
Coches creados sin información: 1
```

En esta práctica, se te proporciona de manera adicional los códigos fuentes finales de estos programas.

3. Creación de un programa concurrente básico en Java: la clase `Thread`

Un programa concurrente se define como un conjunto de procesos secuenciales que interactúan para resolver un problema. Habitualmente, en los lenguajes de programación los procesos secuenciales se implementan mediante hilos, también llamados *threads*. Así, un programa concurrente es en la práctica un conjunto de hilos que interactúan entre sí para la resolución de un problema.

Desde su diseño inicial, Java ya incorporaba la concurrencia como parte del lenguaje. Sin embargo, en las distintas versiones, el modelo de concurrencia ha evolucionado considerablemente, hecho que puede observarse en que algunos métodos que se proporcionaron en el pasado se han descartado en las versiones 1.5 y posteriores (obteniendo el calificativo de *deprecated*). En esta asignatura, vamos a utilizar el modelo de concurrencia actual, correspondiente a la versión 1.7 del lenguaje, desarrollado por un equipo de ingenieros liderado por Doug Lea. Dos referencias importantes sobre ese modelo de concurrencia son la documentación de Java[3] y el libro de Doug Lea [2], en el que se describe cómo integrar la concurrencia con la programación orientada a objetos.

Como se ha explicado anteriormente, Java es un lenguaje orientado a objetos y el mecanismo que se utiliza en Java para declarar *threads* es el habitual en los lenguajes orientados a objetos. En esta sección estudiaremos cómo crear un *thread* en Java y sus aspectos más elementales.

3.1. Construcción y ejecución de un `Thread`

Hemos visto antes que un programa concurrente consta de un conjunto de procesos secuenciales que potencialmente pueden ejecutarse en paralelo. En el modelo de concurrencia de Java, los procesos secuenciales se implementan mediante objetos (instancias) de tipo `Thread`. La clase `Thread` define un número de métodos útiles para su control y gestión (puedes encontrar más información sobre esta clase y sus métodos en la documentación oficial de Java[4]). Entre ellos existen algunos métodos estáticos (palabra clave `static`), que proporcionan información del `Thread` o bien pueden afectar al estado interno del `Thread` que lo invoca. El resto de los métodos disponibles están destinados para que se invoquen desde otros hilos. En las siguientes secciones, se describen los elementos más importantes de un `Thread` en Java.

Cada hilo va a implementarse mediante una instancia (objeto) de la clase `Thread`. A pesar de que hay dos formas de crear un objeto `Thread`, en este curso solo vamos a utilizar una de ellas, tal y como se puede ver en el ejemplo del Código 1.

[3] Disponible en la página web `https://docs.oracle.com/javase/tutorial/essential/concurrency/`.
[4] Disponible en la página web `https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html`

Código 1: Ejemplo de creación de hilos en Java (fichero *HelloRunnable.java*).

```
1  //Fichero HelloRunnable.java
2  public class HelloRunnable implements Runnable {
3      public void run() {
4          System.out.println("Hello from a thread!");
5      }
6
7      public static void main(String args[]) {
8          HelloRunnable miRunnable = new HelloRunnable();
9          Thread miThread = new Thread(miRunnable);
10
11         miThread.start();
12     }
13 }
```

Para crear un Thread vamos a crear una clase que va a implementar la interfaz `Runnable`. Esto se consigue en Java añadiendo `implements Runnable` después de la definición de clase, como puedes observar en la línea 2 del Código 1. Pudes encontrar más información sobre esta interfaz en la documentación de Java[5]. Esta interfaz está compuesta únicamente por el método `run`. Es muy importante destacar que **este método es el que contiene el código del hilo que se ejecutará**. Entre las líneas 3 y 5 del Código 1 está la implementación del método `run`. En este ejemplo, simplemente escribe por pantalla un mensaje.

En el código del método `main`, observa que lo primero que se hace es declarar un objeto de la clase `HelloRunnable` y crearlo, mediante la llamada al constructor (línea 8). Después, en la línea 9 se crea una instancia de `Thread` pasándole como argumento al constructor el objeto de clase `HelloRunnable` creado en la línea anterior.

En este momento, se ha creado la estructura necesaria para mantener un hilo, pero éste todavía no ha empezado a ejecutarse. Su ejecución se inicia con la llamada al método `start()`, realizada en la línea 11 del código. Este método `start()` provocará que internamente se prepare el sistema operativo para lanzar en ejecución el hilo, que acabará llamando eventualmente al método `run()` del objeto de la clase `HelloRunnable` de manera asíncrona.

Es importante resaltar que si se invoca en la línea 11 el método `run()` en lugar de `start()`, la invocación es síncrona y, por tanto, la ejecución es secuencial en lugar de concurrente. Lógicamente, este comportamiento secuencial es el que deseamos evitar en esta asignatura, así que recuerda que **siempre** debes invocar a `start()` en vez de `run()`.

[5] Disponible en la página web `https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html`

Ejercicio 1.

Crea un nuevo proyecto Java en Eclipse y crea un código fuente de nombre `HelloRunnable.java`, copiando en él el código mostrado en el Código 1. Después, modifica el código fuente para que cree 50 hilos y los ponga en ejecución. **¿Podrías modificarla para que, además, cada hilo escriba por pantalla en su mensaje un identificador entero único?**

Consejo: Para resolver este ejercicio, quizás te interese echarle un vistazo al método `getId()` de la clase `Thread`[a].

No hay que entregar nada respecto de este ejercicio.

[a] Consulta la ayuda de Java de la clase `Thread` en la página web `https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html`

3.2. Variables compartidas

En Java, existen varias formas de compartir información almacenada en variables entre hilos. Nosotros vamos a utilizar la forma más general: **vamos a crear un objeto adicional que contenga las variables compartidas y este objeto se pasará como parámetro a los distintos hilos**.

Observa el Código 2, que define la clase `MemoriaCompartida`. Esta clase encapsula (define) una variable de tipo entero (con el identificador `_i`) y con visibilidad privada. Recuerda que esto indica que no se puede acceder a su contenido (es decir, no se puede leer o modificar) desde fuera de la clase. Fíjate que entre las líneas 9 a 11 se define un método `get()`, que devuelve el valor del atributo `_i`, mientras que entre las líneas 13 a 15 se define in método `set(int i)`, que almacena el valor pasado como parámetro en el atributo `_i`. En terminología de POO, estos métodos se denominan *getter* y *setter* del atributo `_i`. En POO, es una norma recomendada que cada clase defina sus atributos como privados, forzando a cualquiera que quiera leer o escribir dichos atributos a usar los métodos de la clase que correspondan. Así, cualquier otra clase que quiera usar `_i` tendrá que hacer uso de su getter o su setter, controlando así siempre el acceso y permitiendo un mejor mantenimiento del código ante eventuales cambios.

Código 2: Ejemplo de compartición de variables entre hilos en Java (fichero *MemoriaCompartida.java*).

```
1  // Fichero MemoriaCompartida.java
2  class MemoriaCompartida {
3      private int _i;
4
5      public MemoriaCompartida (int i){
6          _i = i;
7      }
8
9      public int get(){
10         return _i;
11     }
12
13     public void set(int i){
14         _i = i;
15     }
16 }
```

Observa ahora el Código 3. Este código define la clase `ProcesoContador`, que define un atributo `_n` de la clase `MemoriaCompartida` y un constructor con un parámetro de tipo `MemoriaCompartida`, usado para inicializar el valor del atributo `_n`. Además, esta clase también declara el programa principal, que tiene un pequeño bucle que manipula el atributo `_n`, y el método `run()` (fíjate que de hecho, la clase `ProcesoContador` implementa la interfaz `Runnable`).

Vamos a analizar un poco más qué está haciendo este código del programa principal. Primero, se crea un objeto `n` de tipo `MemoriaCompartida` (línea 19), invocando al constructor de dicha clase con el valor 0 (si te fijas en el Código 2, este constructor colocará el valor de 0 al atributo `_i` del nuevo objeto `n`). Después, se están creando dos objetos de la clase `ProcesoContador`, denominados `count1` y `count2`, invocando al constructor de la clase con el objeto `n` creado anteriormente (líneas 20 y 21). Observa que es en este momento cuando se define que ambos objetos están compartiendo el objeto `n`, dado que se invoca con el mismo elemento para la creación de ambos objetos. Posteriormente, en las líneas 22 y 23 se crean dos objetos `p` y `q` de la clase `Thread` (pasando como argumento al constructor de esta clase los objetos de `ProcesoContador` creados en las líneas anteriores). Tras la creación de estos hilos, se ponen ambos en ejecución mediante la invocación a sendos métodos `start()` (líneas 24 y 25). El programa principal espera a que ambos hilos acaben su ejecución en las líneas 28 y 29, haciendo uso del método `join()` de cada uno de los hilos. Finalmente, el programa principal acaba escribiendo por pantalla el valor del atributo `_i` del objeto `n` invocando a su operación de getter.

Respecto al código anterior, cabe comentar que las invocaciones a los métodos `join()` se encuentran dentro de un entorno `try-catch`. Si consultas la ayuda en línea de Java sobre este método en `https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#join()` verás que aparece definido como se muestra en la Figura 1. Esto indica que el método `join()` de la clase `Thread` es posible que lance una excepción, deno-

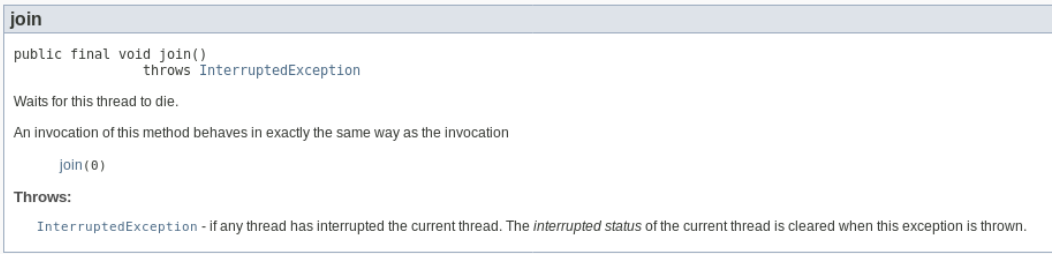
minada `InterruptedException`, y que por tanto, el trozo del código donde se encuentre esta invocación tiene dos opciones:

- **Definir que el método que incluye al código lanza la excepción**. Esto se realiza, de hecho, colocando al final de la definición del método la palabra clave `throws` seguido de la excepción lanzada (en el caso del método `join()`, fíjate que define `throws InterruptedException`). Si ningún método de los llamantes trata la excepción, el programa finalizará con un error de ejecución informando de que se generó una excepción que no fue tratada.
- **Tratar la posible excepción (si ocurre)**. Para tratar una excepción en Java, hay que incluir el código que posiblemente genera la excepción dentro de un bloque `try`. Este bloque `try` va seguido de un bloque `catch(Exception e)`, donde se define qué excepción se quiere calcular y qué acción hay que realizar en caso de que se produzca una excepción. La acción se define dentro del bloque `catch`, mientras que el tipo de excepción capturada se define como parámetro del `catch()`. En la línea 31 del Código 3 se define que se captura la excepción de tipo `InterruptedException` y que en caso de que se produzca, no se hace nada (no hay ningún código dentro del citado bloque `catch`, línea 32).

Ejercicio 2.

En el mismo proyecto Java creado para el Ejercicio 1, crea las clases `MemoriaCompartida` y `ProcesoContador` según se muestra en los Códigos 2 y 3, respectivamente y lanza en ejecución la clase `ProcesoContador`. Estudia el Código 3 y prueba a ejecutar esta clase varias veces para comprobar si siempre obtienes los valores esperados.

No hay que entregar nada respecto de este ejercicio.



join

```
public final void join()
                throws InterruptedException
```

Waits for this thread to die.

An invocation of this method behaves in exactly the same way as the invocation

```
join(0)
```

Throws:

`InterruptedException` - if any thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

Figura 1: Consulta de ayuda en línea de Java para el método `join()` de la clase `Thread`.

Código 3: Ejemplo de compartición de variables entre hilos en Java (fichero *ProcesoContador.java*).

```
1  // Fichero ProcesoContador.java
2  public class ProcesoContador implements Runnable {
3      MemoriaCompartida _n;
4      // el guion bajo es simplemente una notación
5
6      public ProcesoContador (MemoriaCompartida n){
7          _n = n;
8      }
9
10     public void run() {
11         int temp;
12         for (int i = 0; i < 10; i++){
13             temp = _n.get();
14             _n.set(temp + 1);
15         }
16     }
17
18     public static void main(String[] args) {
19         MemoriaCompartida n = new MemoriaCompartida(0);
20         ProcesoContador count1 = new ProcesoContador(n);
21         ProcesoContador count2 = new ProcesoContador(n);
22         Thread p = new Thread(count1);
23         Thread q = new Thread(count2);
24         p.start();
25         q.start();
26
27         try{
28             p.join();
29             q.join();
30         }
31         catch (InterruptedException e){
32             // no hacemos nada en caso de excepción
33         }
34         System.out.println("The value of _i is " + n.get());
35     }
36 }
```

3.3. Otros métodos de la clase `Thread`

Consulta la página web de la documentación de Java sobre la clase `Thread` (página web `https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html`). Aunque son muchos más los métodos disponibles, en esta sección se describen brevemente los más importantes para esta asignatura.

```
public static void sleep(long millis) throws InterruptedException
```

El hilo que invoque el método `sleep()` quedará bloqueado al menos durante `millis` milisegundos. La precisión de esta operación es dependiente del sistema operativo, dado que no todos los sistemas operativos son capaces de tener una precisión de reloj de milisegundos y por tanto puede ser algo variable. Observa que este método es estático, es decir, es un método de clase y no de las instancias de la clase `Thread` (es decir, un hilo que quiera "dormir" durante 100 milisegundos invocará este método como `Thread.sleep(100)`). Al igual que el método `join()`, este método puede lanzar la interrupción `InterruptedException`.

```
public final void join() throws InterruptedException
```

El método `join()` permite a un hilo (el hilo invocante) esperar a la terminación de otro hilo (hilo invocado). Por tanto, si el hilo r está en ejecución y ejecuta el método `join()` de otro hilo t (es decir, ejecuta `t.join()`, la ejecución de r quedará suspendida hasta que el hilo t termine.

```
public static void yield() throws InterruptedException
```

El hilo que invoque este método le indica al scheduler del procesador (componente del procesador que se encarga de gestionar el orden de ejecución de los hilos dentro del procesador) que cede el tiempo de uso del procesador en que se encuentra a otro hilo. Normalmente, el tiempo de ejecución se reparte de manera equitativa entre los hilos que se quieren ejecutar. Con esta llamada, un hilo puede dar parte de su tiempo asignado a otro hilo que se quiera ejecutar. Conviene destacar que el scheduler es libre de ignorar esta indicación (es decir, el hilo que ha llamado este método seguiría su ejecución durante su tiempo asignado).

Este método también sirve como posible heurística para solucionar problemas de inanición, evitando que un hilo monopolice el uso de la CPU mientras que otros no pueden ejecutarse. La utilización de este método debe ser probada exhaustivamente de manera que se garantice que su efecto es el deseado.

4. Concepto de entrelazado

Como se ha descrito al principio de esta práctica, un programa concurrente está compuesto de un conjunto de procesos secuenciales. La programación concurrente asume que, tras la ejecución de una instrucción cualquiera, la siguiente instrucción puede pertenecer a cualquier proceso. Esto representa una abstracción en la ejecución que se conoce como entrelazado arbitrario (en inglés, *arbitrary interleaving*). La construcción de un programa concurrente sobre esta característica permite diseñar programas más robustos e independientes del entorno de ejecución (tanto del hardware como del sistema operativo). El desarrollo de técnicas de programación y de verificación de algoritmos proporcionan mecanismos para asegurar que cualquier entrelazado es correcto y que el programa concurrente realmente está realizando lo deseado.

En concreto, el programas descrito en el Código 3 crea dos hilos que acceden a la misma variable global `n`, que es un objeto de tipo `MemoriaCompartida`. El comportamiento por

defecto de Java al ejecutar este programa es que en casi todas las ejecuciones va a proporcionar como resultado el valor 20, dado que es altamente improbable que haya un cambio de contexto entre los hilos `p` y `q` durante el bucle contenido dentro del método `run()`. Sin embargo, como quizás habrás observado con el ejercicio anterior, es posible que proporcione como resultado un valor diferente de 20.

Ejercicio 3.

Modifica el programa codificado en el ejercicio 2 para provocar de manera artificial cambios de contexto entre las dos instrucciones de asignación dentro del bucle (líneas 13 y 14 del método `run()` de `ProcesoContador`) y visualizar así un comportamiento diferente del esperado. Para ello, intercala (añade) entre ambas líneas una invocación al método `yield()` (recuerda que es estático de la clase `Thread`, con lo que tendrás que añadir en el código la línea `Thread.yield()`). Introduce estos cambios y ejecuta de nuevo el programa para observar las diferencias.

No hay que entregar nada respecto de este ejercicio.

5. Algoritmos para el Problema de la Sección Crítica

En la programación concurrente, el acceso simultáneo a recursos compartidos (por recurso compartido se entiende cualquier dispositivo o componente – como la memoria, el disco duro, la red, etc. – al cual pueden acceder varios procesos a la vez) puede dar lugar a comportamientos inesperados y erróneos (en inglés, *race conditions*). Por tanto, en la programación concurrente, es responsabilidad del programador identificar esas partes del programa concurrente y protegerlas de algún modo para evitar dichos comportamientos anómalos y no deseados. Se denomina *sección crítica de un programa concurrente* a un conjunto de instrucciones que acceden a un recurso compartido. El programa tiene por tanto que **garantizar que en cualquier instante de tiempo no hay más de un proceso accediendo a la sección crítica**, esto es, el acceso a la sección crítica se tiene que realizar en **exclusión mutua**.

Imagina una base de datos donde el acceso a la base de datos para actualizarla no se realizara en exclusión mutua y hubiera multitud de procesos intentando leer o escribir al mismo tiempo en la base de datos. Si no se controla la lectura y escritura, es posible que la información que se almacena se pierda o sea incorrecta, entre otros problemas.

Existen distintos mecanismos para garantizar el acceso en exclusión mutua a la sección crítica. Uno de ellos, aunque exclusivamente de uso académico y con poca utilidad práctica, es el uso de algoritmos de espera activa.

Los algoritmos de espera activa (en inglés, *busy wait*) para el problema de la sección crítica utilizan variables compartidas para sincronizarse, de manera que los procesos concurrentes repetidamente consultan el valor de una de esas variables hasta que este cambia. Cabe notar, por tanto, que consumen CPU sin realizar, en realidad, una computación "útil". Otro de los inconvenientes que presentan es que las variables compartidas

necesarias para la sincronización pueden mezclarse en la declaración con las variables compartidas propias de la sección crítica.

Estos algoritmos funcionan bajo el requisito de que el entorno de ejecución subyacente (el hardware) permite ejecutar instrucciones de carga y escritura de datos de/hacia memoria (operaciones *load* y *store*) de forma atómica. Cabe notar que en general esto no tiene por qué ser así. En Java, una variable de cualquier tipo primitivo, declarada con la palabra clave `volatile` sí que garantiza que estas operaciones son atómicas (en realidad, todos los tipos primitivos excepto los tipos `long` y `double` son `volatile` por defecto).

En los ficheros adjuntos a este guion podéis encontrar los cinco algoritmos de espera activa propuestos por el Prof. Ben-Ari [1] implementados en Java.

Ejercicio 4.

Ejecuta todos los algoritmos de espera activa en Java y comprueba si los resultados del análisis de su corrección que realizamos en la clase de teoría (y que también está disponible en el libro del Prof. Ben-Ari [1]) coincide con las ejecuciones.

No hay que entregar nada respecto de este ejercicio.

Referencias

[1] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice-Hall, Inc., USA, 1990.

[2] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* Addison-Wesley Longman Publishing Co., Inc., USA, 2nd edition, 1999.