UD1 – Concurrencia

Contenido

1 Programas. Procesos. Servicios. Hilos

El día a día requiere cada vez más potencia de cálculo por parte de las aplicaciones informáticas, más y más cálculos numéricos en ingeniería, ciencia, astrofísica, meteorología...

Aunque no es el único factor determinante, los procesadores juegan un papel fundamental a la hora de alcanzar esta potencia. Cada vez son más rápidos y eficientes y los sistemas operativos permiten coordinar varios procesadores para incrementar el número de cálculos por unidad de tiempo. El uso de varios procesadores dentro de un sistema informático se llama multiproceso. Ahora bien, la coordinación de varios procesadores puede provocar situaciones de conflicto que deberán ser gestionadas por el sistema operativo y las aplicaciones.

Un **programa** es un elemento estático, un conjunto de instrucciones, unas líneas de código escritas en un lenguaje de programación, que describen el tratamiento a dar a unos datos iniciales (de entrada) para conseguir el resultado esperado (una salida concreta). En cambio, un **proceso** es dinámico, es una instancia de un programa en ejecución, que realiza los cambios indicados por el programa a los datos iniciales y obtiene una salida concreta. El proceso, además de las instrucciones, requerirá también de recursos específicos para la ejecución como el contador de instrucciones del programa, el contenido de los registros o los datos.

El **sistema operativo** es el encargado de la gestión de procesos. Los crea, los elimina y provee los instrumentos que permiten la ejecución y también la comunicación entre ellos. Cuando se ejecuta un programa, el sistema operativo crea una instancia del programa: el proceso. Si el programa se volviera a ejecutar se crearía una nueva instancia totalmente independiente a la anterior (un nuevo proceso) con sus propias variables, pilas y registros.

Cuando un proceso se encuentra en ejecución se encuentra completamente en memoria y tiene asignados los recursos que necesita. Un proceso no puede escribir en zonas de memoria asignada a otros procesos, la memoria no es compartida. Cada proceso tiene una estructura de datos en la que se guarda la información asociada a la ejecución del proceso. Esta zona se

denomina Bloque de Control de Proceso (BCP). El BCP incluye la siguiente información del proceso:

- Identificador del proceso
- Estado del proceso
- Contador de programa
- Registros de CPU
- Información de planificación de CPU (como la prioridad)
- Información de gestión de memoria
- Tiempo de CPU y tiempo real consumido
- Estado de E/S

Los procesos compiten con otros procesos ejecutados a la vez por los recursos del sistema. Opcionalmente, el sistema operativo permite también que los procesos puedan comunicarse y colaborar entre ellos.

En Windows obtenemos la lista de procesos en el sistema operativo con el comando ***tasklist***.

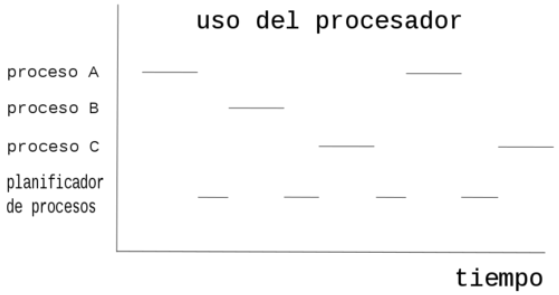
Con el comando ***ps*** de Linux podemos ver parte de la información asociada a cada proceso. La ejecución del comando con los parámetros -f o -AF mostrará información adicional.

Hasta ahora hemos hablado de elementos de software, tales como programas y procesos, pero no del hardware utilizado para ejecutarlos. El elemento de hardware en el que se ejecutan los procesos es el procesador. Un **procesador** es el componente de hardware de un sistema informático encargado de ejecutar las instrucciones y procesar los datos. Cuando un dispositivo informático está formado por un único procesador hablaremos de sistemas monoprocesador, por el contrario, si está formado por más de un procesador hablaremos de sistemas multiprocesador.

Actualmente, la mayoría de los sistemas operativos aprovechan los tiempos de reposo de los procesos (cuando esperan, por ejemplo, algún dato del usuario o se encuentran pendientes de alguna operación de entrada / salida) para introducir el procesador otro proceso, simulando así una ejecución paralela.

De forma genérica llamaremos los procesos que se ejecuten a la vez, ya sea de forma real o simulada, **procesos concurrentes**. La ejecución de procesos concurrentes, aunque sea de forma simulada, hace aumentar el rendimiento del sistema informático ya que aprovecha más el tiempo del procesador. Es el sistema operativo el encargado de gestionar la ejecución concurrente de diferentes procesos contra un mismo procesador.

Hablamos de **multiprogramación** cuando el sistema operativo gestiona la ejecución de procesos concurrentes a un sistema monoprocesador. La siguiente figura muestra como el tiempo de CPU es repartido entre varios procesos.

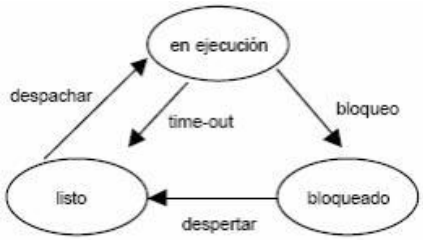


Los sistemas informáticos monoprocesador actuales tienen una gran velocidad y si han de ejecutar más de un proceso a la vez irán alternando su ejecución simulando un multiproceso. Cuando en un ordenador monoprocesador aplican técnicas de multiprogramación los beneficios en rendimiento no son tan evidentes como en sistemas informáticos multiprocesadores.

En un sistema informático multiprocesador existen dos o más procesadores, por lo tanto, se pueden ejecutar simultáneamente varios procesos. También se pueden calificar de multiprocesadores aquellos dispositivos el procesador de los cuales tiene más de un núcleo, es decir, tienen más de una CPU en el mismo circuito integrado del procesador. Evidentemente la arquitectura es diferente, pero este sistema, al igual que un equipo multiprocesador, nos permite ejecutar de manera simultánea procesos diferentes.

1.1 Estados de un proceso

Los sistemas operativos disponen de un planificador de procesos encargado de repartir el uso del procesador de la forma más eficiente posible y asegurando que todos los procesos se ejecuten en algún momento. Para realizar la planificación, el sistema operativo se basará en el estado de los procesos para saber qué necesitarán el uso del procesador. Los procesos en disposición de ser ejecutados organizarán en una cola esperando su turno.



El primer estado sería nuevo, es decir, cuando un proceso es creado. Una vez creado, pasa al estado de **listo o preparado**. En este momento el proceso está preparado para hacer uso del procesador, está compitiendo por el recurso del procesador. El planificador de procesos del sistema operativo es el que decide cuando entra el proceso a ejecutarse. Cuando el proceso se está ejecutado, su estado se denomina **en ejecución**. De nuevo, es el planificador el encargado de decidir cuándo abandona el procesador.

Cuando un proceso abandone el procesador cambiará al estado de preparado y se quedará esperando que el planificador le asigne el turno para conseguir de nuevo el procesador y volver al estado de ejecución.

Desde el estado de ejecución, un proceso puede pasar al estado de **bloqueado** o en espera. En este estado, el proceso estará a la espera de un evento, como puede ser una operación de entrada / salida, o la espera de la finalización de otro proceso, etc. Cuando el evento esperado suceda, el proceso volverá al estado de preparado, guardando el turno con el resto de procesos preparados.

El último estado sería el de **terminado**. Es un estado al que se llega una vez el proceso ha finalizado toda su ejecución y estará listo para que el sistema libere cuando pueda los recursos asociados.

Las transiciones del estado de un proceso pueden ser provocadas por alguno de los motivos siguientes:

- De ejecución a bloqueado: el proceso realiza alguna operación de entrada y salida o el proceso debe esperar que otro proceso modifique algún dato o libere algún recurso que necesita.
- De ejecución a preparado: el sistema operativo decide sacar un proceso del procesador porque ha sido un tiempo excesivo haciendo uso y lo pasa al estado de preparado. Asigna a otro proceso el acceso al procesador.
- De preparado a ejecución: es el planificador de procesos, dependiendo de la política que practique, el encargado de hacer este cambio.
- De bloqueado a preparado: el recurso o el dato por la que había sido bloqueado está disponible o ha terminado la operación de entrada y salida.
- De ejecución en acabado: el proceso finaliza sus operaciones o bien otro proceso o el sistema operativo lo hacen terminar.

Cada vez que un proceso cambia al estado de ejecución, se denomina **cambio de contexto**, porque el sistema operativo debe almacenar los resultados parciales alcanzados durante la ejecución del proceso saliendo y hay que cargue los resultados parciales de la última ejecución del proceso entrando en los registros del procesador, asegurando la accesibilidad a las variables y el código que toque seguir ejecutando.

Un **servicio** es un tipo de proceso que no tiene interfaz con el usuario. Pueden ser, dependiendo de su configuración, inicializado por el sistema de forma automática, en el que realizando sus funciones sin que el usuario se entere o bien se pueden mantener a la espera de que alguien les haga una petición para hacer una tarea en concreto.

Dependiendo de cómo se están ejecutando, los procesos se clasificarán como procesos en primer plano (*foreground*, en inglés) y procesos en segundo plano (*background*). Los procesos en primer plano mantienen una comunicación con el usuario, ya sea informando de las acciones realizadas o esperando sus peticiones por medio de alguna interfaz de usuario. En cambio, los que se ejecutan en segundo plano no se muestran explícitamente al usuario, bien porque no necesitan su intervención o bien porque los datos requeridos no se incorporan a través de una interfaz de usuario.

El nombre *deamon* proviene de las siglas inglesas DAEMON (*Disk And Execution Monitor*). A menudo en la literatura técnica se ha extendido este concepto usando la palabra "demonio", por lo que puede encontrar esta palabra haciendo referencia a los programas que se ejecutan en segundo plano.

Los servicios son procesos ejecutados en segundo plano. *Servicio* es una nomenclatura utilizada en Windows. En sistemas Linux se denominan también procesos *deamon*.

Como los servicios no disponen de interfaz de usuario directa almacenan la información generada o los posibles errores que vayan produciendo en ficheros de registro habitualmente conocidos como *logs*.

Los servicios pueden estar en ejecución local, en nuestro ordenador, como el Firewall, el antivirus, la conexión Wi-Fi, o los procesos que controlan nuestro ordenador, o bien pueden ejecutarse en algún ordenador de un sistema informático distribuido o en Internet, etc. Las llamadas a los servicios distribuidos se realizan sobre protocolos de comunicación.

Por ejemplo, los servicios de http, DNS (Domain Name System), FTP (File Transfer Protocol) son servicios que nos ofrecen servidores que están en Internet.

Un **hilo** es un concepto más avanzado que un proceso: al hablar de procesos cada uno tiene su propio espacio en memoria. Si abrimos 20 procesos cada uno de ellos consume 20x de memoria RAM. Un hilo es un proceso mucho más ligero, en el que el código y los datos se comparten de una forma distinta.

Un proceso no tiene acceso a los datos de otros procesos. Sin embargo, un hilo sí accede a los datos de otro hilo. Esto complicará algunas cuestiones a la hora de programar.

Los procesos son llamados **entidades pesadas** porque están en espacios de direccionamiento de memoria independientes, de creación y de comunicación entre procesos, lo que consume muchos recursos de procesador. En cambio, ni la creación de hilos ni la comunicación consumen mucho tiempo de procesador. De ahí que los hilos se denominan **entidades ligeras**.

Un proceso estará en ejecución mientras alguno de sus hilos esté activo. Sin embargo, también es cierto que, si finalizamos un proceso de forma forzada, sus hilos también terminarán la ejecución.

Podemos hablar de dos niveles de hilos: los hilos de nivel de usuario, que son los que creamos cuando programamos con un lenguaje de programación como Java; y los hilos de segundo nivel que son los que crea el sistema operativo para apoyar a los primeros. Nosotros programaremos nuestros hilos utilizando unas librerías que nos proporciona el lenguaje de programación. Son las librerías con las instrucciones pertinentes que indicarán al sistema los hilos que han de crear y cómo gestionarlos.

En referencia al procesamiento, cada hilo se procesa de forma independiente, aunque pertenezca o no a un mismo proceso. La única diferencia entre el tratamiento de hilos y procesos la encontramos a nivel de memoria. Para hilos de un mismo proceso el sistema operativo debe mantener los mismos datos en memoria. Para hilos de diferentes procesos en cambio hay que disponer de datos independientes. En caso de que sea necesario que un mismo procesador alterne la ejecución de diversos procesos, habrá restaurar la memoria específica del proceso en cada cambio, esto se denomina también cambio de contexto. Si la alternancia de procesamiento se hace entre hilos de un mismo proceso no hará falta restaurar la memoria (cambiar el contexto) y, por lo tanto, el cambio de ejecución entre hilos del mismo proceso será más eficiente que el cambio entre procesos en el procesador.

2 Creación de procesos con Java

Java dispone de varias clases en el paquete *java.lang* para la gestión de procesos:

La clase *ProcessBuilder* permite crear procesos del sistema operativo.

Los métodos ProcessBuilder.start () y Runtime.exec crean un proceso nativo y devuelven una instancia de una subclase de *Process* que se puede usar para controlar el proceso y obtener información sobre éste. La clase *Process* proporciona métodos para realizar la entrada del proceso, realizar la salida al proceso, esperar a que el proceso se complete, verificar el estado de salida del proceso y destruir (kill) el proceso.

El método *waitFor* de la clase *Process* indica al hilo en ejecución que debe esperar a que el subproceso representado por la clase termine.

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class LaunchCalc {

    public static void main(String[] args) {
        try {
            // Lanzamos la calculadora creado un nuevo proceso en el SO de ese programa
            Process pCalc = new ProcessBuilder("calc").start();
            // Mostrar por consola el PID del proceso lanzado
            System.out.println("Lanzado proceso calculadora con java. PID: " + pCalc.pid());
            // vamos a lanzar el comando tasklist para ver los procesos lanzados en el SO
            // Comando a ejecutar
            String command = "tasklist";
            // Lanzamos el proceso que permite ejecutar el comando anterior
            Process pCommand = Runtime.getRuntime().exec(command);
            // Accedemos al buffer de lectura del proceso lanzado
            BufferedReader reader = new BufferedReader(new InputStreamReader(pCommand.getInputStream()));
            // leemos el resultado del comando
            // y mostramos por consola el proceso que corresponde con el PID del proceso
            // anterior
            String line = "";
            while ((line = reader.readLine()) != null) {
                if (line.contains(Long.toString(pCalc.pid()))) {
                    System.out.println(line + "\n");
                }
            }
            // Esperamos a que el proceso del comando termine
            pCommand.waitFor();
            // Esperamos a que el proceso calc finalice
            pCalc.waitFor();
            // Mostrar que el programa va a finalizar
            System.out.println("Programa finalizado");
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Veamos un ejemplo de ejecución de comandos de consola de Windows (cmd) a través de ProcessBuilder:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class LaunchCmd {

    public static void main(String[] args) {
        try {
            // Lanzamos el proceso que permite ejecutar comandos cmd
            ProcessBuilder pb = new ProcessBuilder("cmd.exe", "/c", "ping www.iesch.org");
            Process p = pb.start();

            // Accedemos al buffer de lectura del proceso lanzado
            BufferedReader reader = new BufferedReader(new InputStreamReader(p.getInputStream()));

            // leemos el resultado de los comandos
            // y lo mostramos por consola
            String line = "";
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }

            // Esperamos a que el proceso termine
            p.waitFor();
            System.out.println("Fin de la ejecución");

        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

En el siguiente ejemplo vemos cómo ejecutar una clase java en un subproceso de un programa java. Primero crearemos una clase que realiza la suma de los números comprendidos en un intervalo y muestra el resultado por consola.

```
public class IntervalAddition {
    public static void main(String[] args) {

        // crea un objeto de tipo IntervalAddition para hacer la suma
        IntervalAddition addition = new IntervalAddition();
        // Recupera los parámetros recibidos
        int n1 = Integer.parseInt(args[0]);
        int n2 = Integer.parseInt(args[1]);
        // obtiene el resultado
        int res = addition.add(n1, n2);
        // Lo muestra por la salida
        System.out.println("Addition of interval " + n1 + " to " + n2 + " is " + res);
        System.out.flush();// vacía el buffer de salida

    }

    // Suma los números del intervalo [n1,n2]
    public int add(int n1, int n2) {
        int res = 0;
        for (int i = n1; i <= n2; i++) {
            res += i;
        }
        return res;
    }
}
```

A continuación, creamos una clase lanzador que lance subprocesos que ejecutan la clase anterior. Para ello necesitaremos especificar la ruta donde se encuentra la clase y el nombre completo de esta. Para ver por la consola del proceso principal la salida de los subprocesos hijos deberemos redirigir la salida de los procesos hijos hacia el padre.

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.lang.ProcessBuilder.Redirect;

public class IntervalAditionLauncher {

	public static void main(String[] args) {
		try {
			//Nuevo objeto lanzador
			IntervalAditionLauncher launcher = new IntervalAditionLauncher();
			//Lanzamos un par de intervalos para su suma
			launcher.launch(1, 3);
			launcher.launch(1, 10);

		} catch (Exception e) {
			e.printStackTrace();
		}

	}

	public void launch(Integer n1, Integer n2) {
		//Muestra por consola el inicio del proceso
		System.out.println("Start " + n1 + " to " + n2);
		//Obtenemos la ruta dónde se encuentra la clase
		String classPath = System.getProperty("java.class.path");
		//Nombre completo incluyendo el paquete de la clase que realiza la suma
		String className ="ud1.ejercicios.IntervalAddition";
		//Creamos un nuevo proceso que ejecutará la clase de la suma
		//Redirigimos la salida del proceso hijo al proceso padre
		java.lang.ProcessBuilder pb = new java.lang.ProcessBuilder("java", "-cp",
classPath, className, n1.toString(), n2.toString()).redirectOutput(Redirect.INHERIT);
		try {
			//Iniciamos el proceso
			pb.start();

		} catch (Exception e) {
			e.printStackTrace();
		}
	}

}
```

3 Concurrencia

Actualmente, una gran cantidad de aplicaciones utilizan la programación concurrente. En los sistemas operativos actuales existen muchas aplicaciones ejecutándose al mismo tiempo y compartiendo información. Por ejemplo, podemos escribir en un editor de texto, tener puesto un reproductor de música y descargar un vídeo al mismo tiempo. También podemos tener en ejecución un navegador capaz de cargar en varias pestañas diferentes páginas web. Son ejemplos que ilustran la idea de programación concurrente.

Gracias a la evolución que ha sufrido el hardware durante los últimos años se han creado sistemas operativos que pueden optimizar los recursos de los procesadores y pueden ejecutar diferentes procesos de forma simultánea. La ejecución simultánea de procesos se denomina también **concurrencia**. No quiere decir que se deban ejecutar exactamente en el mismo momento, el intercalado de procesos también se considera ejecución concurrente. La mayoría de los lenguajes de programación actuales pueden hacer uso de este recurso y diseñar aplicaciones en las que los procesos se ejecuten de manera concurrente.

Hablamos de **programación concurrente** cuando se ejecutan en un dispositivo informático de forma simultánea diferentes tareas (procesos).

Si la programación concurrente se da en un computador con un único procesador hablaremos de **multiprogramación**. En cambio, si el computador tiene más de un procesador y, por tanto, los procesos se pueden ejecutar de forma realmente simultánea, hablaremos de **programación paralela**.

En un dispositivo multiprocesador la concurrencia es real, los procesos son ejecutados de forma simultánea en diferentes procesadores del sistema. Es habitual que el número de procesos que se esté ejecutando de forma concurrente sea mayor que el número de procesadores. Por lo tanto, será obligatorio que algunos procesos se ejecuten sobre el mismo procesador.

3.1 Ventajas

Hemos definido programación concurrente o concurrencia como la técnica por la que múltiples procesos se ejecutan al mismo tiempo y pueden comunicarse entre ellos. La mayoría de los sistemas intentan aprovechar esta concurrencia para incrementar la capacidad de ejecución.

Incrementar la potencia de cálculo y el rendimiento es uno de las principales ventajas. Cuando se ejecutan varios procesos al mismo tiempo, la velocidad de ejecución global se puede incrementar. Hay que tener en cuenta, sin embargo, que no siempre es así. A veces, dependiendo de la complejidad de la aplicación, las técnicas de sincronismo o comunicación son más costosas en tiempo que la ejecución de los procesos.

Un sistema multiprocesador es **flexible**, ya que, si aumentan los procesos que se están ejecutando es capaz de distribuir la carga de trabajo de los procesadores, y puede también reasignar dinámicamente los recursos de memoria y los dispositivos para ser más eficientes. Es de **fácil crecimiento o escalable**. Si el sistema lo permite, se pueden añadir nuevos procesadores de forma sencilla y así aumentan su potencia.

Los sistemas multiprocesador pueden ser sistemas redundantes. El hecho de disponer de varios procesadores, puede permitir un aumento de la disponibilidad de los recursos (más procesadores al servicio del usuario) o bien el uso de procesadores especializados en tareas de

verificación y control. En el último caso hablaremos de sistemas con una alta **tolerancia a fallos**. Un fallo de un procesador no hace que el sistema se detenga.

Los sistemas multiproceso nos permiten diferenciar procesos por su **especialización** y, por tanto, reservar procesadores para operaciones complejas, aprovechar a otros para procesamientos paralelos y para adelantar la ejecución.

3.2 Inconvenientes

Exclusión mutua. En programación concurrente es muy típico que varios procesos accedan a la vez a una variable compartida para actualizarla. Esto se debe evitar, ya que puede producir inconsistencia de datos. Uno puede estar actualizando la variable a la vez que otro la puede estar leyendo. Por ello es necesario conseguir la exclusión mutua de los procesos respecto a la variable compartida. Para ello se propuso la región crítica, que sería el trozo de código de un proceso que puede interferir con otro proceso. Cuando dos o más procesos comparten una variable, el acceso a dicha variable debe efectuarse siempre dentro de una región crítica asociada a la variable. Solo uno de los procesos podrá acceder para actualizarla y los demás deberán esperar. El tiempo de estancia es finito.

Condición de sincronización. Hace referencia a la necesidad de coordinar los procesos con el fin de sincronizar sus actividades. Puede ocurrir que un proceso P1 llegue a un estado X que no pueda continuar su ejecución hasta que otro proceso P2 haya llegado a un estado Y de su ejecución. La programación concurrente proporciona mecanismos para bloquear procesos a la espera de que ocurra un evento y para desbloquearlos cuando este ocurra.

Algunas herramientas para manejar la concurrencia son: la región crítica, los semáforos, buzones, eventos o monitores entre otros.

4 Programación paralela

Cuando la programación concurrente se realiza en un sistema multiprocesador hablamos de **programación paralela**.

En los ordenadores multiprocesadores la concurrencia es real, por lo tanto, el tiempo de ejecución suele ser menor que en dispositivos monoprocesador.

La principal desventaja de la programación paralela son los controles que tenemos que añadir para que la comunicación y sincronización de los procesos que se ejecutan concurrentemente sean correctos. Dos procesos se deben sincronizar o comunicar cuando un proceso necesita algún dato que está procesando el otro o bien uno de ellos tiene que esperar la finalización del otro para poder continuar su ejecución.

Java tiene integrado el apoyo a la concurrencia en el propio lenguaje como veremos a lo largo de esta unidad.

No siempre se podrán paralelizar las tareas de un sistema; en primer lugar, porque hay aplicaciones cuya ejecución debe ser secuencial para obtener los resultados esperados. Y, en segundo lugar, por la dificultad para desarrollar entornos de programación en sistemas paralelos.

Se han generalizado los sistemas informáticos multinúcleo a servidores, ordenadores de sobremesa y también en sistemas móviles, *smartphones*, tabletas, etc. Todos estos dispositivos ayudan a potenciar la programación concurrente y paralela. Pero la programación paralela requiere la sincronización de procesos y el control de los datos compartidos, lo que añade complejidad a la programación.

La programación paralela está muy ligada a la arquitectura del sistema de computación. Pero cuando programamos en lenguajes de alto nivel debemos abstraernos de la plataforma en la que se ejecutará. Los lenguajes de alto nivel nos proveen de librerías que nos facilitan esta abstracción y nos permiten utilizar las mismas operaciones para programar de forma paralela que acabarán implementadas para usarse en una plataforma concreta.

4.1 Ventajas

- Proporciona ejecución simultánea de tareas.
- Disminuye el tiempo total de ejecución de una aplicación.
- Resolución de problemas complejos y de grandes dimensiones.
- Utilización de recursos no locales, por ejemplo, los recursos que están en una red distribuida, una WAN o la propia red Internet.
- Disminución de costos, en vez de gastar en un supercomputador muy caro se pueden utilizar otros recursos más baratos disponibles remotamente.

4.2 Inconvenientes

- Los compiladores y entornos de programación para sistemas paralelos son más difíciles de desarrollar.
- Los programas paralelos son más difíciles de escribir.
- Mayor complejidad en el acceso a los datos.
- El consumo de energía de los elementos que forman el sistema.
- La comunicación y la sincronización entre diferentes subtareas.

5 Programación distribuida

Un tipo especial de programación paralela es la llamada **programación distribuida**. Esta programación se da en sistemas informáticos distribuidos. Un sistema distribuido está formado por un conjunto de ordenadores que pueden estar situados en lugares geográficos diferentes unidos entre ellos a través de una red de comunicaciones. Un ejemplo de sistema distribuido puede ser el de un banco con muchas oficinas en el mundo, con un ordenador central para oficina para guardar las cuentas locales y hacer las transacciones locales. Este equipo puede comunicarse con los otros ordenadores centrales de la red de oficinas. Cuando se hace una transacción no importa donde se encuentra la cuenta o el cliente.

La **programación distribuida** es un tipo de programación concurrente en la que los procesos son ejecutados en una red de procesadores autónomos o en un sistema distribuido. Es un sistema de computadores independientes que desde el punto de vista del usuario del sistema se ve como una sola computadora.

En los sistemas distribuidos, a diferencia de los sistemas paralelos, no existe memoria compartida, por lo tanto, si necesitan la utilización de variables compartidas se crearán réplicas de estas variables en los diferentes dispositivos de la red y habrá que controlar la coherencia de los datos.

La comunicación entre procesos para intercambiar datos o sincronizarse entre ellos se hace con mensajes que se envían a través de la red de comunicaciones que comparten.

La programación distribuida es un paradigma de programación enfocado a desarrollar sistemas distribuidos, abiertos, escalables, transparentes y tolerantes a fallos. Es el resultado natural del uso de las computadoras y las redes.

Una arquitectura típica es la arquitectura cliente-servidor.

El cloud computing o computación en la nube es un ejemplo de programación distribuida.

Existen varios modelos de programación para la comunicación entre procesos en un sistema distribuido, por ejemplo:

- **Sockets**. Proporcionan los puntos externos para la comunicación entre procesos. Es actualmente la base de la comunicación. Pero al ser de muy bajo nivel de abstracción, son adecuados a nivel de aplicación. Posteriormente veremos su tratamiento en Java.
- Llamada de procedimientos remotos o **RPC** (Remote Procedure Call). Permite a un programa cliente llamar a un procedimiento de otro programa en ejecución en un proceso servidor. El proceso servidor define en su interfaz de servicio los procedimientos disponibles para ser llamados remotamente.
- **Invocación remota de objetos**. El modelo de programación basado en objetos ha sido extendido para permitir que los objetos de diferentes procesos se comuniquen uno con otro por medio de una invocación a un método remoto o RMI (Remote Method Invocation). Un objeto que vive en un proceso puede invocar métodos de un objeto que reside en otros procesos. Java RMI extiende el modelo de objetos de Java para proporcionar soporte de objetos distribuidos en lenguaje Java.

5.1 Ventajas

- Se puede compartir recursos y datos.
- Capacidad de crecimiento incremental.
- Mayor flexibilidad al poderse distribuir la carga de trabajo entre diferentes ordenadores.
- Alta disponibilidad, tolerancia a fallos.
- Soporte de aplicaciones inherentemente distribuidas.
- Carácter abierto y heterogéneo.

5.2 Inconvenientes

- Aumento de la complejidad, se necesita un nuevo tipo de software.
- Problemas con las redes de comunicación: pérdida de mensajes, saturación del tráfico.
- Problemas de seguridad como por ejemplo ataques de denegación de servicio en la que se bombardea un servicio con peticiones inútiles de forma que un usuario interesado en usarlo no pueda emplearlo.

6 Hilos

La programación multihilo permite llevar a cabo diferentes hilos de ejecución a la vez, es decir, permite realizar diferentes tareas en una aplicación de forma concurrente. La mayoría de lenguajes de programación permiten trabajar con hilos y realizar programación multihilo. También son multihilo la mayoría de las aplicaciones que usamos en nuestros ordenadores (editores de texto, navegadores, editores gráficos ...), lo que nos da la sensación de más agilidad de ejecución.

En un editor de texto, por ejemplo, un hilo puede estar controlando la ortografía del texto, otro puede estar atento a las pulsaciones sobre los iconos de la interfaz gráfica y el otro guardando el documento.

La programación tradicional está diseñada para trabajar de forma secuencial, de modo que cuando termina un proceso se pone en marcha otro. Los hilos son una forma de ejecutar paralelamente diferentes partes de un mismo programa. En Java los hilos son conocidos como *threads*.

Un hilo o *thread* es la unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo.

Del mismo modo que un sistema operativo permite ejecutar diferentes procesos a la vez por concurrencia o paralelismo, dentro de un proceso habrá uno o varios hilos en ejecución.

Los hilos representan exclusivamente las ejecuciones de las instrucciones de un programa que se llevan a cabo simultáneamente en el contexto de un mismo proceso. Es decir, compartiendo el acceso a la misma zona de memoria asignada al proceso al que pertenecen. Cada proceso contiene al menos un hilo, a pesar de que puede llegar a contener muchos.

Cabe recordar que los procesos no comparten memoria entre ellos, son independientes, contienen información sobre su estado e interactúan con otros procesos a través de mecanismos de comunicación dados por el sistema. En cambio, los hilos comparten recursos. Esto hace que cualquier hilo pueda modificar los recursos compartidos. Si un hilo modifica un dato en la memoria, el resto de los hilos tiene acceso al nuevo dato de forma inmediata. Cuando hay un proceso ejecutándose, como mínimo siempre hay un hilo en ejecución. Hablamos de procesos multihilo cuando se ejecutan varios hilos concurrentemente, realizando diferentes tareas y colaborando entre ellos.

Cuando un proceso finaliza, todos sus hilos también lo hacen. De forma equivalente, cuando finalizan todos los hilos de un proceso, el proceso también termina y todos sus recursos son liberados.

6.1 Gestión de hilos

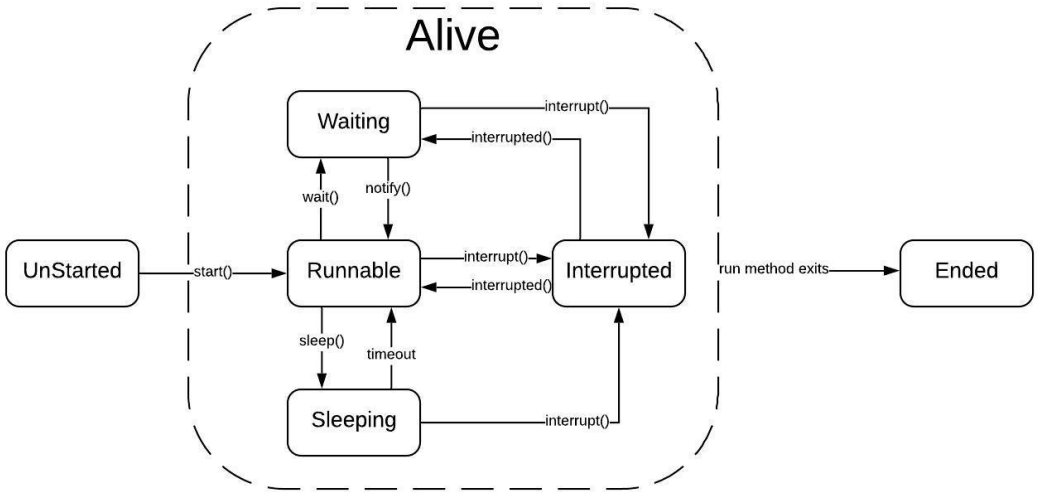
Cuando arrancamos un programa en Java hay un hilo en ejecución, el **hilo principal**, que es creado por el método *main*. Este hilo es importante ya que es el encargado, si es necesario, de crear el resto de hilos del programa. Se debe programar la aplicación para que el hilo principal sea el último en terminar su ejecución. Esto se consigue haciendo esperar los hilos creados por el hilo principal que este último finalice.

Si no se crean más hilos que el principal, sólo existe uno y será el encargado de hacer las llamadas a los métodos y crear objetos que indique el programa.

Cada hilo está asociado con una instancia de la clase *Thread*. Hay dos estrategias básicas para usar objetos *Thread* para crear una aplicación concurrente:

- Controlar directamente la creación y administración de subprocesos, simplemente creando una instancia de tipo *Thread* cada vez que la aplicación necesite iniciar una tarea asincrónica.
- Abstraer la gestión de subprocesos del resto de su aplicación, pasando las tareas de la aplicación a un *executor*.

Veamos en la siguiente figura los estados de los hilos en java y sus transiciones.



Veremos ahora el uso y la gestión de objetos de tipo *Thread*. Los ejecutores se estudiarán con otros objetos en el apartado Concurrencia de alto nivel en Java.

6.1.1 Definir e iniciar hilos

Una aplicación que crea una instancia de *Thread* debe proporcionar el código que se ejecutará en ese hilo. Hay dos maneras de hacer esto:

- Proporcionando un objeto *Runnable*. La interfaz *Runnable* define un único método, run, que contendrá el código ejecutado en el hilo. El objeto *Runnable* debe ser pasado al constructor de la clase *Thread.*

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        Thread t = new Thread(new HelloRunnable());
        t.start();
    }

}
```

- Creando una subclase de *Thread*. La clase *Thread* ya implementa la interfaz *Runnable*, aunque el método *run* no hace nada. Se deberá implementar dicho método en la clase hija.

```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

Se debe llamar el método *start* para iniciar la ejecución del nuevo hilo.

La primera implementación es más general, ya que un objeto que implementa una interfaz puede heredar de cualquier otra clase, mientras que en el segundo caso estamos forzados a heredar de la clase *Thread*. La primera aproximación, además de ser más flexible, se puede aplicar a las soluciones de alto nivel que veremos más adelante.

6.1.2 Pausar la ejecución de un hilo con *Sleep*

El método *sleep* suspende la ejecución de hilo actual para un periodo de tiempo especificado. Es una manera eficiente de hacer que el tiempo del procesador esté disponible para los otros subprocesos de una aplicación u otras aplicaciones que podrían estar ejecutándose en un sistema informático. El método *sleep* también se puede usar para ajustar el ritmo de ejecución, como se muestra en el ejemplo que sigue, y esperar a la ejecución de otro hilo cuya tarea se espere en un periodo de tiempo.

El siguiente ejemplo muestra como el hilo se detiene 3 segundos después de imprimir cada mensaje.

```
public class SleepMessages implements Runnable {

	public void run() {
		// mensajes
		String importantInfo[] = { "Programas", "Procesos"
				,"Servicios", "Hilos" };

		for (int i = 0; i < importantInfo.length; i++) {
			// Mostrar mensaje
			System.out.println(importantInfo[i]);

			try {
				// Pausar 3 segundos
				Thread.sleep(3000);
			} catch (InterruptedException e) {
				// Mostrar interrupción
				System.out.println("Hilo interrumpido");
			}

		}
		System.out.println("***Hilo finalizado***");
	}

	public static void main(String[] args) throws InterruptedException {
		// Crear nuevo hilo
		Thread t = new Thread(new SleepMessages());
		// Arrancar hilo
		t.start();
	}
}
```

El método *sleep* lanza la excepción *InterruptedException* cuando otro hilo interrumpe la ejecución de éste mientras *sleep* está activo.

El método *sleep* proporciona dos versiones sobrecargadas: una que especifica el tiempo de suspensión en milisegundos y otra que especifica el tiempo de suspensión en nanosegundos. Sin embargo, no se garantiza que estos tiempos de *sleep* sean precisos, ya que están limitados por las el sistema operativo. Además, el periodo de suspensión puede ser interrumpido por interrupciones, como veremos en la sección siguiente. En cualquier caso, no puede suponer que la invocación *sleep* suspenderá el hilo exactamente durante el periodo de tiempo especificado.

6.1.3 Interrupciones

El método *interrupt* es una indicación a un hilo para que este pare lo que esté haciendo para hacer otra cosa. El programador debe decidir cómo responder a la interrupción, aunque es muy común que el hilo termine. Para que el mecanismo de interrupción funcione correctamente, el hilo interrumpido debe implementar su propia interrupción. Veamos los siguientes ejemplos:

El hilo interrumpido se detiene

```
public class SleepMessages implements Runnable {

	public void run() {
		// mensajes
		String importantInfo[] = { "Programas", "Procesos"
				, "Servicios", "Hilos" };

		for (int i = 0; i < importantInfo.length; i++) {
			// Mostrar mensaje
			System.out.println(importantInfo[i]);

			try {
				// Pausar 3 segundos
				Thread.sleep(3000);
			} catch (InterruptedException e) {
				// Mostrar interrupción
				System.out.println("Hilo interrumpido");

				// Finalizar la ejecución del hilo
				return;
			}

		}
		System.out.println("***Hilo finalizado***");
	}

	public static void main(String[] args) throws InterruptedException {
		// Crear nuevo hilo
		Thread t = new Thread(new SleepMessages());
		// Arrancar hilo
		t.start();
		//dejar pasar 4 segundos e interrumpir el hilo
		Thread.sleep(4000);
		t.interrupt();
	}

}
```

En el siguiente ejemplo el hilo interrumpido no finaliza a pesar de la interrupción

```
public class SleepMessages implements Runnable {

	public void run() {
		// mensajes
		String importantInfo[] = { "Programas", "Procesos"
			, "Servicios", "Hilos" };

		for (int i = 0; i < importantInfo.length; i++) {
			// Mostrar mensaje
			System.out.println(importantInfo[i]);

			try {
				// Pausar 3 segundos
				Thread.sleep(3000);
			} catch (InterruptedException e) {
				// Mostrar interrupción
				System.out.println("Hilo interrumpido..." + e);
				//Al capturar la interrupción y no responder
				//la ejecución del hilo continúa
			}

		}
		System.out.println("***Hilo finalizado***");
	}

	public static void main(String[] args) throws InterruptedException {
		// Crear nuevo hilo
		Thread t = new Thread(new SleepMessages());
		// Arrancar hilo
		t.start();
		//dejar pasar 4 segundos e interrumpir el hilo
		Thread.sleep(4000);
		t.interrupt();
	}
}
```

En el siguiente ejemplo el hilo interrumpido se comporta normalmente ya que éste no se encuentra dormido o en espera. Lo que si ocurre es que el hilo queda marcado como interrumpido.

```
public class SleepMessages implements Runnable {

	public void run() {
		// mensajes
		String importantInfo[] = { "Programas", "Procesos"
			, "Servicios", "Hilos" };

		for (int j = 0; j < 100; j++) {
			for (int i = 0; i < importantInfo.length; i++) {
				// Mostrar mensaje
				System.out.println(importantInfo[i]);
			}
		}
		System.out.println("***Hilo finalizado***");
	}

	public static void main(String[] args) throws InterruptedException {
		// Crear nuevo hilo
		Thread t = new Thread(new SleepMessages());
		// Arrancar hilo
		t.start();
		// interrumpir el hilo
		Thread.sleep(5);
		t.interrupt();
		// mostrar el flag interrupted del hilo
		System.out.println("Hilo " + t.getName() + " interrumpido: "
			+ t.isInterrupted());
	}
}
```

Otra opción podría ser responder a la interrupción del hilo haciendo que el método *run* lance una excepción de tipo *InterruptedException*. De forma que el hilo se detenga y otra parte del código se encargue de manejar esa excepción.

6.1.4 Joins

El método *join* permite a un hilo esperar a la finalización de otro hilo. La ejecución de la siguiente línea hace que el hilo actual pause su ejecución hasta que el hilo *t* termine.

```
t.join();
```

Igual que *sleep*, *join* responde a una interrupción con una excepción de tipo *InterruptedException.*

6.2 Gestión de hilos por parte del sistema operativo

La máquina virtual de Java (JVM) es un sistema multihilo, y es la encargada de gestionar los hilos, de asignar el tiempo de ejecución, las prioridades, etc. Es la encargada de decidir qué hilo entra a ejecutarse. La gestión que hace de los hilos es similar a la que tiene el sistema operativo cuando gestiona más de un proceso. La máquina virtual de Java es un proceso dentro del sistema operativo, es decir, ya que los hilos se ejecutan en la máquina virtual, comparten recursos.

6.2.1 Planificación de hilos

La manera que tiene el sistema operativo de comportarse con la ejecución de hilos es un tema importante en la programación multihilo. La planificación se refiere a qué política seguir para decidir qué hilo toma el control del procesador y en qué momento. También se ha de planificar

cuándo debe dejar de ejecutarse. Java está diseñado para que sea un sistema portable. El sistema de hilos debe ser soportado por cualquier plataforma y, ya que cada plataforma funciona de forma diferente, el tratamiento de los hilos debe ser muy general. Como características más importantes de la planificación de hilos tenemos:

- Todos los hilos tienen prioridad y el encargado de decidir qué hilo ejecuta debe garantizar que los hilos con prioridad más alta tengan preferencia. Pero esto no implica que en un momento dado un hilo con prioridad más baja esté ejecutándose.
- Se debe garantizar que todos los hilos se ejecuten en algún momento.
- El tiempo asignado a cada hilo para hacer uso del procesador puede aplicarse o no dependiendo del sistema operativo en el que se ejecuta la máquina virtual.

Atendiendo a estas premisas los hilos podrán intercalar su ejecución en cualquier momento con comportamientos que serán impredecibles. Debemos tener esto en cuenta y si queremos establecer un determinado orden en la ejecución de hilos, deberemos añadir código en nuestro programa para conseguirlo.

6.2.2 Prioridad de hilos

Las prioridades de los hilos en Java son inicialmente la misma que los hilos creadores. Por defecto tienen una prioridad de 5. El rango de prioridades va desde 1, definida con la propiedad de *Thread MIN_PRIORITY*, hasta 10, definida por *MAX_PRIORITY*. La prioridad por defecto con valor 5 es *NORM_PRIORITY*.

Los hilos con una prioridad más alta serán puestos a ejecución por parte del planificador en primer lugar y los de prioridad más baja cuando los de prioridad superior estén bloqueados o hayan terminado la ejecución. La clase *Thread* tiene dos métodos que permiten ver y modificar las prioridades de los hilos:

- *setPriority()* permite modificar la prioridad de un hilo.
- *getPriority()* devuelve la prioridad de un hilo.

Los hilos de las clases que generan interfaces de usuario (Swing y AWT) tienen por defecto una prioridad de 6. De esta manera la interacción con el usuario es más fluida.

El método *yield()* hace que un hilo que se está ejecutando pase de nuevo al estado y a la cola de preparados. De esta manera permite que otros hilos de prioridad superior o de la misma prioridad se puedan ejecutar. Pero puede suceder que una vez pasado a preparado vuelva a ser puesto en ejecución de nuevo.

Veamos un ejemplo, podremos comprobar que no siempre el hilo con más prioridad es el que se ejecuta antes.

```
public class SamplePriorities extends Thread {

	public SamplePriorities(String name) {
		this.setName(name);
	}

	public void run() {
		// mostrar información por consola
		System.out.println("Executing [" + this.getName() + "]");
		for (int i = 0; i < 5; i++) {
			System.out.println("\t(" + this.getName() + ": " + i + ")");
		}
	}

	public static void main(String[] args) {
		// crear hilos
		SamplePriorities h1 = new SamplePriorities("One");
		SamplePriorities h2 = new SamplePriorities("Two");
		SamplePriorities h3 = new SamplePriorities("Three");
		SamplePriorities h4 = new SamplePriorities("Four");
		SamplePriorities h5 = new SamplePriorities("Five");
		// establecer prioridades
		h1.setPriority(MIN_PRIORITY);
		h2.setPriority(3);
		h3.setPriority(NORM_PRIORITY);
		h4.setPriority(7);
		h5.setPriority(MAX_PRIORITY);
		// ejecución
		h1.start();
		h2.start();
		h3.start();
		h4.start();
		h5.start();
	}

}
```

A la hora de programar hilos con prioridades hemos de tener cuenta que el comportamiento no está garantizado y dependerá de la plataforma en la que se ejecute el programa. En la práctica casi nunca hay que establecer prioridades de manera manual.

6.2.3 Hilos demonios

Cuando creamos un hilo y antes de llamar al método *start()* podemos indicar que el hilo será ejecutado como un demonio (*daemon* en inglés). Esto permitirá al hilo ejecutarse en un segundo plano. Los hilos demonios tienen la prioridad más baja. Si un hilo demonio crea otro hilo, este también será un demonio. Esta cualidad se hereda. Y no se puede cambiar una vez el hilo ha sido inicializado.

Garbage collector es un demonio creado por la máquina virtual de Java, que se encarga de liberar memoria ocupada por objetos que no están referenciados.

Para indicar que un hilo es un demonio utiliza el método *setDaemon(true)*. En el caso de *setDaemon(false)*, el hilo es de usuario que es la opción por defecto. Los hilos demonios finalizan cuando finaliza el último hilo de usuario y la aplicación termina.

Para ver si un hilo es un demonio, se utiliza el método *isDaemon()* que devolverá un booleano indicando si el hilo es un demonio o no.

7 Sincronización

Los hilos se comunican principalmente compartiendo el acceso a campos y los objetos a los cuales esos campos se refieren. Esta forma de comunicación es extremadamente eficiente, pero crea dos tipos posibles de error: interferencia entre hilos y errores de inconsistencia de memoria. La manera de prevenir estos errores es la **sincronización**.

Sin embargo, la sincronización puede ocasionar contención en la ejecución de los hilos cuando dos o más hilos intentan acceder el mismo recurso simultáneamente y causar que la máquina virtual de Java ejecute uno o más hilos más lentamente, o incluso suspender su ejecución. La inanición (starvation) y el interbloqueo (deadlock, livelock) son formas de contención de ejecución de hilos.

Llamamos ***interbloqueo*** a la situación extrema que nos encontramos cuando dos o más procesos están esperando la ejecución del otro para poder continuar de manera que nunca lograrán desbloquearse.

Llamamos ***inanición* (starvation)** a la situación que se produce cuando un proceso no puede continuar su ejecución por falta de recursos.

7.1 Interferencia entre hilos

Veamos el siguiente ejemplo de contador

```
public class Counter {

    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int getValue() {
        return c;
    }
}
```

La siguiente clase lanza 2 hilos en paralelo que actualizan el valor del contador de forma repetida

```
public class LanzadorContador implements Runnable{

	private int value = 0;
	private static Counter counter = new Counter();

	public LanzadorContador(int value) {
		super();
		this.value = value;
	}

	public static void main(String[] args) {
		try {
			Thread t1 = new Thread(new LanzadorContador(100000));
			Thread t2 = new Thread(new LanzadorContador(-100000));
			t1.start();
			t2.start();
			t1.join();
			t2.join();

			// resultado esperado: 0
			System.out.println(counter.getValue());
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}

	@Override
	public void run() {

		if(value>=0) {
			for (int i = 0; i < value; i++) {
				counter.increment();
			}
		} else {
			for (int i = 0; i > value; i--) {
				counter.decrement();
			}
		}
	}

}
```

Al ejecutarse el programa ambos hilos van a interferir y no tenemos garantizado que la salida del programa sea cero que es lo que cabría esperar. Incluso que parece que las operaciones que hacen son sentencias simples *c++* o *c--*, en realidad, se pueden descomponer en 3 operaciones:

1. Obtener c
2. Aumentar (o disminuir) el valor en 1.
3. Guardar el nuevo valor en c.

Cuando dos hilos obtienen el valor al mismo tiempo y cada uno guarda el valor resultante en la variable hay incrementos y decrementos que se perderán. En otras ocasiones puede ocurrir que lo hagan correctamente, el resultado será impredecible. Como no ocurre ningún error de ejecución estas situaciones pueden ser difíciles de detectar y corregir.

7.2 Errores de consistencia de memoria

Los errores de consistencia de memoria ocurren cuando diferentes hilos tienen lecturas inconsistentes de lo que debería ser el mismo dato. Las causas de los errores de inconsistencia de memoria son complejos y van más allá del alcance de este curso. Afortunadamente, no necesitamos una comprensión detallada de estas causas, lo que necesitamos conocer es una estrategia para evitarlos.

La clave para evitar los errores de consistencia de memoria es entender la relación de lo que sucede-antes (*happens-before*). Esta relación es simplemente una garantía que la memoria escrita por una sentencia específica es visible por otra sentencia específica.

Para ver esto, consideramos el siguiente ejemplo:

```
int counter = 0;
```

La variable counter es compartida entre dos hilos, A y B. Supongamos que el hilo A incrementa counter.

```
counter++;
```

Entonces enseguida el hilo B imprime counter:

```
System.out.println(counter);
```

Si las dos sentencias han sido ejecutadas en el mismo hilo, sería seguro asumir que el valor impreso podría ser “1”. Pero si las dos sentencias son ejecutadas en hilos separados, el valor impreso podría ser “0”, porque no hay garantía que el cambio hecho por el hilo A sobre counter sea visible al hilo B, a menos que el programador haya establecido una relación *happens-before* entre estas dos sentencias.

Hay muchas acciones que crean relaciones *happens-before*. Una de ellas es la sincronización, como vamos a ver a lo largo de la unidad didáctica.

Ya hemos visto dos acciones que nos permiten asegurar el orden de ejecución

- Cuando se invoca *Thread.start* en un fragmento de código, todas las sentencias anteriores a esta de dicho fragmento ocurren antes que cada una de las sentencias a ejecutar por el hilo que se arranca. Por lo tanto, los efectos del código que preceden a la creación del nuevo hilo son visibles para el nuevo hilo.
- Cuando un hilo termina y causa un *Thread.join* en otro hilo al que vuelve, entonces todas las sentencias ejecutadas por el hilo terminado tienen una relación *happens-before* con todas las sentencias que posteriores al *join*. Los efectos del código en el hilo son ahora visibles para el hilo que ejecuta el *join*.

Podemos encontrar más información en java.util.concurrent.

7.3 Sección crítica

Un mecanismo utilizado para la solución a los problemas que acabamos de ver pasará por obligar a acceder a los recursos a través de la ejecución de un código que llamaremos **sección crítica** y que nos permitirá proteger aquellos recursos con mecanismos que impidan la ejecución simultánea de dos o más hilos dentro de los límites de la sección crítica.

Este algoritmo de sincronización que evita el acceso a una región crítica para más de un hilo o proceso y que garantiza que únicamente un proceso estará haciendo uso de este recurso y el

resto que quieran utilizarlo estarán a la espera hasta que sea liberado, se denomina **algoritmo de exclusión mutua**.

Exclusión mutua (mutex, *mutual exclusion*) es el tipo de sincronización que impide que dos procesos ejecuten simultáneamente una misma sección crítica.

Las instrucciones que forman parte de una sección crítica se ejecutarán con si fueran una única instrucción. Se deben sincronizar los procesos para que un único proceso o hilo pueda excluir de forma temporal en el resto de procesos de un recurso compartido (memoria, dispositivos, etc.) de tal forma que la integridad del sistema quede garantizada.

En Java, la palabra reservada ***synchronized*** permite aplicar exclusión mutua (mutex) a métodos y fragmentos de código.

Todos los métodos con el modificador *synchronized* se ejecutarán en exclusión mutua. El modificador asegura que si se está ejecutando un método sincronizado ningún otro método sincronizado se pueda ejecutar. Pero los métodos no sincronizados pueden estar ejecutándose y con más de un proceso a la vez. Además, únicamente el método sincronizado puede estar ejecutado por un proceso, el resto de procesos que quieren ejecutar el método deberán esperar que termine el proceso que lo está ejecutando.

Los fragmentos de código marcados en Java con la palabra clave *synchronized* se consideran secciones críticas.

Veamos cómo podemos conseguir que nuestro contador funcione de manera correcta:

```
public class SynchronizedCounter {

    private int c = 0;

    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int getValue() {
        return c;
    }
}
```

Utilizamos la nueva clase *SynchronizedCounter* para nuestro contador

```
public class LauncherSyncCounter implements Runnable{

	private int value = 0;
	private static SynchronizedCounter counter
		= new SynchronizedCounter();

	public LauncherSyncCounter(int value) {
		super();
		this.value = value;
	}

	public static void main(String[] args) {
		try {
			Thread t1 = new Thread(new LanzadorContador(100000));
			Thread t2 = new Thread(new LanzadorContador(-100000));
			t1.start();
			t2.start();
			t1.join();
			t2.join();

			// resultado esperado: 0
			System.out.println(counter.getValue());
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}

	@Override
	public void run() {

		if(value>=0) {
			for (int i = 0; i < value; i++) {
				counter.increment();
			}
		} else {
			for (int i = 0; i > value; i--) {
				counter.decrement();
			}
		}
	}

}
```

Al ejecutar ahora nuestro código siempre obtenemos el valor esperado: cero. Hacer que los métodos de la clase *SynchronizedCounter* sean *synchronized* tiene dos consecuencias:

- Primero, no es posible que dos invocaciones de métodos sincronizados en el mismo objeto que se solapen. Cuando un hilo está ejecutando un método sincronizado para un objeto, todos los otros hilos que invocan métodos sincronizados para el mismo objeto se bloquean (suspenden la ejecución) hasta que el primero hilo termine con el objeto.
- Segundo, cuando un método sincronizado termina, este automáticamente establece una relación happens-before con cualquier invocación posterior de un método sincronizado para el mismo objeto. Esto garantiza que los cambios de estado del objeto sean visibles para todos los hilos.

Los constructores no se pueden marcar como *synchronized*, tratar de hacerlo genera un error de sintaxis. No tiene sentido sincronizar un constructor ya que únicamente el hilo que lo ejecuta tiene acceso al objeto mientras lo está construyendo.

Advertencia. Cuando se construye un objeto que va a ser compartido entre hilos, hay que muy cuidadoso que no existan "leaks" de la referencia al objeto de manera prematura. Por ejemplo, supongamos que queremos mantener en una lista llamada *instances* el conteniendo cada instancia de la clase. Si incluimos la siguiente línea en el constructor

```
instances.add(this);
```

nos encontraremos con que otros hilos podrán usar la colección *instances* para acceder al objeto antes que la construcción del objeto esté completa.

Los métodos sincronizados ofrecen una estrategia simple para evitar errores de consistencia de memoria e interferencia de hilos: si un objeto es visible por más de un hilo, bastará con que todas las lecturas y escrituras de las variables del objeto se hagan a través de métodos sincronizados.

7.4 Bloqueos intrínsecos y sincronización

La sincronización está implementada entorno a una entidad interna conocida como bloqueo intrínseco (*intrinsic lock)* o *monitor lock* (o simplemente "monitor"). Los bloqueos intrínsecos juegan un rol en ambos aspectos de sincronización: forzando el acceso exclusivo a los estados de un objeto y estableciendo relaciones *happens-before* que son esenciales para la visibilidad.

Cada objeto tiene un bloqueo intrínseco asociado él. Por convención, un hilo que necesita acceso exclusivo y consistente a los campos de un objeto y tiene que adquirir el bloqueo intrínseco del objeto antes de acceder a él y liberarlo cuando haya terminado con él. Un hilo se dice que mantiene el bloqueo intrínseco entre el momento en que este es adquirido y en el momento en que es liberado. Tan pronto como el hilo obtiene un bloqueo intrínseco, ningún otro hilo puede adquirir el mismo bloqueo. Los demás hilos se bloquearán cuando intenten obtener el bloqueo.

Cuando un hilo libera un bloqueo intrínseco, una relación happens-before es establecida entre esa acción y cualquier obtención subsecuente del mismo bloqueo.

Cuando un hilo invoca un método sincronizado, automáticamente adquiere un bloqueo intrínseco del objeto al que pertenece el método y lo libera cuando el método termina. La liberación del bloqueo ocurre también si se sale del método a causa de una excepción no controlada.

Los métodos estáticos están asociados a la clase y no a una instancia del objeto en particular. Veamos qué sucede en estos casos; cuando un método estático sincronizado es invocado, entonces el bloqueo intrínseco se produce asociado a la clase, esto es, el acceso a los campos estáticos está controlado por un bloqueo intrínseco distinto del de cualquier instancia de la clase.

Bloques sincronizados

Otra forma para crear código sincronizado es con bloques sincronizados. A diferencia de los métodos sincronizados, los bloques sincronizados deben especificar el objeto que provee el bloqueo intrínseco:

```
public void addName(String name) {
   synchronized(this) {
       lastName = name;
       nameCount++;
```

```
    }
    nameList.add(name);
}
```

En este ejemplo, el método *addName* necesita sincronizar los cambios a las variables *lastName* y *nameCount*, pero también necesita evitar invocaciones sincronizadas de los otros métodos del objeto. Sin bloques sincronizados, debería haber un método sincronizado separado para actualizar las variables y otro no sinctronizado con el único propósito de invocar *nameList.add*.

Los bloques sincronizados son también útiles para mejorar la concurrencia a un nivel más fino de sincronización. Supongamos, por ejemplo, la clase *MsLunch* que tiene dos atributos de instancia, *c1* y *c2*, que nunca son usados juntos. Todas las actualizaciones a estos atributos deben ser sincronizadas, pero no hay razón para evitar una actualización de *c1* sea intercalada con una actualización de *c2*, y haciendo eso reduce la concurrencia creando un bloqueo innecesario. En lugar de usar métodos sincronizados o a cambio de usar bloqueos asociados con *this*, creamos dos objetos únicamente para obtener los bloqueos.

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Hay que ser extremadamente cuidadoso con esta característica. Debemos estar absolutamente seguros de que es realmente seguro intercalar acceso de los atributos en discusión.

Sincronización Reentrante

Recordemos que un hilo no puede adquirir el bloqueo que tiene otro hilo. Pero un hilo puede obtener un bloqueo que él ya tiene. Permitiendo al hilo adquirir el mismo bloqueo más de una vez permite la sincronización reentrante. Esto se corresponde con una situación donde el código sincronizado, directa o indirectamente, invoca un método que también contiene código sincronizado, y ambos conjuntos de código usan el mismo bloqueo. Sin sincronización reentrante, el código sincronizado tendría que tomar muchas precauciones adicionales para evitar que un hilo se bloquee a sí mismo.

7.5 Acceso atómico

En programación, decimos que una acción es atómica si sucede toda de una sola vez. Una acción atómica no puede ejecutarse a medias: se ejecuta completamente o no se ejecuta en

absoluto. No hay efectos colaterales visibles de una acción atómica hasta que la acción está completa.

Ya hemos visto la expresión de incremento, *c++*, no es una acción atómica. Incluso expresiones muy simples pueden definir acciones complejas que se pueden descomponer en otras acciones. Sin embargo, hay acciones que sí son atómicas:

- Lecturas y escrituras para variables por referencia y para la mayoría de variables primitivas (todas excepto long y double).
- Lecturas y escrituras son atómicas para todas las variables declaradas *volatile* (incluidas las variables long y double.).

Las acciones atómicas no pueden ser solapadas, así que se pueden usar sin temor a interferencia entre hilos. Sin embargo, esto no elimina toda la necesidad de sincronizar acciones atómicas, ya que todavía son posibles errores de consistencia de memoria. Usar variables volátiles reduce el riesgo de errores de consistencia de memoria, porque cualquier escritura a una variable *volatile* establece una relación happens-before con lecturas posteriores de la misma variable. Esto significa que cambios en la variable *volatile* son siempre visibles a los otros hilos. Es más, esto también significa que cuando un hilo lee una variable volátil, éste ve no sólo el último cambio hecho al valor de la variable *volatile*, sino también los efectos colaterales del código que llevó a cabo el cambio.

Usar simples accesos atómicos a una variable es más eficiente que acceder a estas variables a través de código sincronizado, pero requiere tener más cuidado para evitar errores de consistencia de memoria. Si el esfuerzo adicional vale la pena, depende del caso y la complejidad de la aplicación.

Algunas de las clases en el paquete java.util.concurrent ofrecen métodos atómicos que no usan sincronización. Se tratarán en el apartado Concurrencia de alto nivel en Java.

7.6 Problemas comunes

7.6.1 Deadlocks

Deadlock describe una situación en la que dos o más hilos están bloqueados para siempre, esperando uno al otro entre sí.

Veamos un ejemplo: Pedro y Pablo son amigos y muy respetuosos con las reglas de cortesía. Una regla estricta de cortesía es que cuando saludas a un amigo, esperas hasta que tu amigo tiene la oportunidad de devolverte el saludo. Desafortunadamente, esta regla no ha contado con la posibilidad de que ambos amigos puedan saludarse al mismo tiempo. Este ejemplo, *Deadlock*, modela esta posibilidad:

```
public class Deadlock {

	static class Friend {
		private final String name;

		public Friend(String name) {
			this.name = name;
		}

		public String getName() {
			return this.name;
		}

		public synchronized void bow(Friend bower) {
			System.out.format("%s: %s" + "  has bowed to me!%n"
				, this.name, bower.getName());
			bower.bowBack(this);
		}

		public synchronized void bowBack(Friend bower) {
			System.out.format("%s: %s" + " has bowed back to me!%n"
				, this.name, bower.getName());
		}
	}

	public static void main(String[] args) {
		final Friend pedro = new Friend("Pedro");
		final Friend pablo = new Friend("Pablo");
		// arrancamos el saludo de pedro a plablo
		// ojo: clase anónima
		new Thread(new Runnable() {
			public void run() {
				pedro.bow(pablo);
			}
		}).start();
		new Thread(new Runnable() {
			public void run() {
				pablo.bow(pedro);
			}
		}).start();
	}
}
```

Cuando *Deadlock* se ejecuta, es extremadamente probable que los dos hilos se bloqueen cuando intentan invocar la devolución del saludo, *bowBack*. Ninguno de los dos bloqueos finaliza nunca porque cada hilo está esperando al otro para salir del saludo, *bow*.

Veamos una forma de resolver el deadlock; en lugar de tener los métodos de saludo sincronizados, lo que hace que se bloqueen los objetos pedro y pablo esperando cada uno por la respuesta del otro, bloqueamos ambos objetos cuando vamos a saludar sincronizando el código que hace la llamada al método que efectivamente hace el saludo (*performBow*) especificando el bloqueo de ambos objetos (el que saluda y el saludado). Se ordenan los objetos por el hash de manera que se acceda al bloqueo intrínseco en orden y evitar así un posible deadlock. Ningún método queda declarado *synchronized*.

```
public class Deadlock {

	static class Friend {
		private final String name;


		public Friend(String name) {
			this.name = name;
		}

		public String getName() {
			return this.name;
		}

		public void bow(Friend bower) {
			// Ordenar los objetos para realizar el objeto de 2 objetos
			// siempre en el mismo orden evitando de esta forma el deadlock
			Friend first, second;
			if(System.identityHashCode(this)<System.identityHashCode(bower)) {
				first = this;
				second = bower;
			} else {
				first = bower;
				second = this;
			}
			// Sincronizar el saludo bloqueando ambos objetos
			synchronized (first) {
				synchronized (second) {
					System.out.format("%s has bowed to me!%n"
						, this.name, bower.getName());
					bower.bowBack(this);
				}
			}
		}

		public synchronized void bowBack(Friend bower) {
			System.out.format("%s has bowed back to me!%n"
				, this.name, bower.getName());
		}

	}

	public static void main(String[] args) {
		final Friend pedro = new Friend("Pedro");
		final Friend pablo = new Friend("Pablo");
		// arrancamos el saludo de pedro a plablo
		// ojo: clase anónima
		new Thread(new Runnable() {
			public void run() {
				pedro.bow(pablo);
			}
		}).start();
		new Thread(new Runnable() {
			public void run() {
				pablo.bow(pedro);
			}
		}).start();
	}
}
```

7.6.2 Inanición

La inanición (starvation) describe la situación en la que un hilo es incapaz de obtener acceso de forma regular a los recursos comunes y por lo tanto no es capaz de progresar. Esto sucede cuando los recursos comunes están bloqueados durante largos periodos de tiempo por hilos "glotones"; si un hilo tiene largos periodos de ejecución y éste se ejecuta durante mucho tiempo y accede frecuentemente y de forma sincronizada al mismo objeto el resultado será que este objeto queda bloqueado casi todo el tiempo

7.6.3 Livelock

Un hilo a veces actúa en respuesta a la acción de otro hilo. Si la acción del otro hilo es también una respuesta a la acción de otro hilo distinto, entonces esta situación puede derivar en un livelock. Como el deadlock, los hilos livelocked no pueden seguir avanzando. Sin embargo, los hilos no están bloqueados, simplemente están muy ocupados respondiendo el uno al otro impidiendo la ejecución de la tarea. Esto es comparable a dos personas que intentan cruzarse en un pasillo estrecho: Pedro se mueve a su izquierda para permitir a Pablo pasar, mientras Pablo se mueve a su derecha para permitir a Pedro pasar. Como vuelven a bloquearse el paso, entonces Pedro se mueve a su derecha para permitir a Pablo pasar, mientras Pablo se mueve a su izquierda y continúan bloqueándose entre sí y así continuamente.

La inanición y los livelocks son problemas que aparecen de manera mucho menos frecuente que los deadlocks.

7.7 Guarded blocks en Java

Los hilos a menudo tienen que coordinar sus acciones. El estilo de coordinación más común se conoce como **guarded block**. Un bloque de este tipo comienza verificando una condición que debe ser verdadera antes que el bloque pueda continuar. Hay un número de pasos para seguir para que se haga correctamente, veámoslo.

Supongamos, por ejemplo, que el método *guardedJoy* que no debe iniciar hasta que la variable compartida *joy* ha sido establecida por otro hilo. Ese método podría, en teoría, simplemente iterar hasta que la condición sea satisfecha pero esa iteración es un desperdicio, ya que estará continuamente ejecutándose mientras espera.

```
public void guardedJoy() {
   // Simple loop guard. Wastes
   // processor time. Don't do this!
   while(!joy) {}
   System.out.println("Joy has been achieved!");
}
```

Una forma más eficiente de guarda invoca *Object.wait* para suspender el hilo actual. La invocación de *wait* no retorna hasta que otro hilo ha enviado una notificación de que un evento especial pudo haber ocurrido, así que no tiene por qué ser necesariamente el evento que el hilo está esperando.

```
public synchronized void guardedJoy() {
    // This guard only loops once for each special event, which may not
    // be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
```

```
    System.out.println("Joy and efficiency have been achieved!");
}
```

Nota: Siempre tenemos que invocar *wait* dentro de un bucle que verifica la condición por la que estamos esperando. No debemos asumir que la interrupción fue por una condición particular por la que estamos esperando o que dicha condición está aún en true.

Como muchos métodos que suspenden la ejecución, *wait* puede lanzar *InterruptedException*. En este ejemplo, podemos ignorar esa excepción y vigilar únicamente el valor de *joy*.

¿Por qué está sincronizada esta versión de guardedJoy? Supongamos que *obj* es el objeto que estamos usando para invocar *wait*. Cuando un hilo invoca *obj.wait*, este debe obtener el bloqueo intrínseco para *obj*, de otro modo un se lanza un error. Invocar *wait* dentro de un método sincronizado es una manera simple para obtener el bloqueo intrínseco.

Cuando *wait* es invocado, el hilo libera el bloqueo y suspende la ejecución. En algún momento futuro, otro hilo va a obtener el mismo bloqueo e invocar *Object.notifyAll*, informando a todos los hilos que están esperando por ese bloqueo de que algo importante acaba de pasar:

```
public synchronized void notifyJoy() {
   joy = true;
   notifyAll();
}
```

Algún tiempo después que el segundo hilo ha liberado el bloqueo, el primer hilo vuelve a adquirir el bloqueo y vuelve retomando desde la invocación del *wait*.

Nota: hay un segundo método de notificación, *notify*, el cual despierta un único hilo. Ya que el método *notify* no permite especificar el hilo que debe despertar, es útil sólo en aplicaciones masivamente paralelas, esto es, programas con un número grande de hilos, todos haciendo tareas similares. En este tipo de aplicaciones daría igual que hilo ha despertado.

7.8 Objetos Inmutables

Un objeto es considerado **inmutable** si su estado no puede ser cambiado después de ser creado. Es una estrategia ampliamente aceptada para crear código simple y confiable.

Los objetos inmutables son particularmente útiles en aplicaciones concurrentes. Ya que no pueden cambiar de estado, no pueden terminar en estados inconsistentes o corruptos por la interferencia de hilos.

Tradicionalmente, los programadores somos reacios a utilizar objetos inmutables, suele preocupar el coste de crear un objeto nuevo frente a actualizar un objeto existente. Se sobrestima habitualmente el impacto de la creación de objetos y este impacto puede ser compensado las eficiencias asociadas a los objetos inmutables. Entre estas eficiencias encontramos la disminución de la sobrecarga debido al funcionamiento del Garbage Collector y la eliminación del código necesario para proteger objetos mutables de la corrupción.

En el siguiente ejemplo vemos qué estrategias seguir para definir objetos inmutables frente al uso de una clase que utiliza sincronización.

La clase *SynchronizedRGB* define objectos que representan colores. Cada objecto representa el color como tres enteros que contienen los valores de los tres colores primarios y una cadena de caracteres que da nombre al color.

```
public class SynchronizedRGB {

    // Values must be between 0 and 255.
    private int red;
    private int green;
    private int blue;
    private String name;

    private void check(int red,
                       int green,
                       int blue) {
        if (red < 0 || red > 255
            || green < 0 || green > 255
            || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public SynchronizedRGB(int red,
                           int green,
                           int blue,
                           String name) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }

    public void set(int red,
                    int green,
                    int blue,
                    String name) {
        check(red, green, blue);
        synchronized (this) {
            this.red = red;
            this.green = green;
            this.blue = blue;
            this.name = name;
        }
    }

    public synchronized int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }

    public synchronized String getName() {
        return name;
    }

    public synchronized void invert() {
        red = 255 - red;
        green = 255 - green;
        blue = 255 - blue;
        name = "Inverse of " + name;
    }
}
```

Esta clase debe usarse con cuidado para evitar que entre en un estado inconsistente. Supongamos, por ejemplo, que un hilo ejecuta el siguiente código:

```
SynchronizedRGB color = new SynchronizedRGB(0, 0, 0, "Pitch Black");
```

```
...
int myColorInt = color.getRGB();      //Sentencia 1
String myColorName = color.getName(); //Sentencia 2
```

Si otro hilo invoca el método *color.set* después de la *Sentencia 1* y antes de la *Sentencia 2* resultará que *myColorInt* y *myColorName* tendrán valores inconsistentes. Podríamos evitar este problema ejecutando ambas sentencias de manera atómica:

```
synchronized (color) {
    int myColorInt = color.getRGB();
    String myColorName = color.getName();
}
```

Este tipo de inconsistencias sólo pueden aparecer en objetos mutables, pero nunca serán un problema con una versión inmutables de *SynchronizedRGB*.

Veamos qué reglas tenemos que seguir para crear objetos inmutables:

1. No se deben implementar métodos “setter”, métodos que modifiquen atributos u objetos referidos por atributos.
2. Todos los atributos se definirán como *final* y *private*.
3. No permitir que las subclases sobrescriban métodos. La manera más simple de conseguir esto es definir la clase como *final*. Una forma más sofisticada es hacer el constructor privado y construir las instancias en métodos *factory*.
4. Si los atributos de instancia incluyen referencias a objetos mutables, no permitir que estos objetos puedan ser cambiados:
    a. No proporcionar métodos que modifiquen los objetos mutables.
    b. No compartir referencias a los objetos mutables. Nunca se almacenarán referencias a objetos mutables externos pasados al constructor; si es necesario, crear copias y almacenar referencias a las copias. De manera análoga, crear copias de los objetos internos mutables cuando sea necesario evitando así devolver referencias a los objetos originales en los métodos de la clase.

Para aplicar esta aproximación a la clase *SynchronizedRGB* requerirá los siguientes pasos:

1. El método *set* no tiene cabida en una clase inmutable, debe eliminarse. Si necesitamos un objeto color diferente crearemos un nuevo objeto.
2. El método *invert* se cambiará para devolver una instancia de un nuevo objeto inmutable en lugar de modificar el objeto existente.
3. Todos los atributos además de tener el modificador *private* tendrán el modificador *final*.
4. La propia clase de definirá como *final*.
5. Como el único atributo que se refiere a un objeto es de tipo String que es un objeto inmutable, ya no tenemos que tener ninguna precaución en este sentido.

```
final public class ImmutableRGB {

    // Values must be between 0 and 255.
    final private int red;
    final private int green;
    final private int blue;
    final private String name;

    private void check(int red,
                       int green,
                       int blue) {
        if (red < 0 || red > 255
            || green < 0 || green > 255
            || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public ImmutableRGB(int red,
                        int green,
                        int blue,
                        String name) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }


    public int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }

    public String getName() {
        return name;
    }

    public ImmutableRGB invert() {
        return new ImmutableRGB(255 - red,
                       255 - green,
                       255 - blue,
                       "Inverse of " + name);
    }
}
```

7.9 Patrones

A lo largo de la historia se han propuesto soluciones específicas para los problemas de sincronización que hemos visto que vale la pena estudiar a pesar de que resulta difícil generalizar una solución ya que dependen de la complejidad, la cantidad de secciones críticas, del número de procesos que requieran exclusión mutua y de la interdependencia existente entre procesos.

Para validar cualquier mecanismo de sincronización de una sección crítica se deben cumplir los siguientes criterios:

- Exclusión mutua: no puede haber más de un proceso simultáneamente en la sección crítica.
- No inanición (starvation): un proceso no puede esperar un tiempo indefinido para entrar a ejecutar la sección crítica.

- No interbloqueo (deadlock, livelock): ningún proceso de fuera de la sección crítica puede impedir que otro proceso entre en la sección crítica.
- Independencia del hardware: inicialmente no se deben hacer suposiciones respecto al número de procesadores o la velocidad de los procesos.

Semáforos, monitores y paso de mensajes son los patrones más usados para resolver los problemas de sincronización. Veamos en qué consisten los dos primeros.

7.9.1 Semáforos

Imaginemos una carretera de un solo carril que tiene que pasar por un túnel. Hay un semáforo en cada extremo del túnel que nos indica cuando podemos pasar y cuando no. Si el semáforo está en verde el coche pasará de forma inmediata y el semáforo pasará a rojo hasta que salga. Este símil nos introduce en la definición real de un semáforo.

Los semáforos son una técnica de sincronización de memoria compartida que impide la entrada del proceso en la sección crítica todo bloqueándolo. El concepto fue introducido por el informático holandés Dijkstra para resolver el problema la exclusión mutua y permitir resolver gran parte de los problemas de sincronización entre procesos.

Los semáforos, no sólo controlan el acceso a la sección crítica, sino que además disponen de información complementaria para poder decidir si es necesario o no bloquear el acceso de aquellos procesos que lo soliciten. Así, por ejemplo, serviría para solucionar problemas sencillos con poca interdependencia.

De forma genérica distinguiremos 3 tipos de operaciones en un semáforo:

El semáforo admite 3 operaciones:

- Inicializar: se trata de la operación que permite poner en marcha el semáforo. La operación puede recibir un valor por parámetro que indicará si éste comenzará boqueando (rojo) o liberado (verde).
- Liberar: cambia el valor interno del semáforo poniéndolo en verde (la libera). Si existen procesos en espera, los activa para que finalicen su ejecución.
- Bloquear: sirve para indicar que el proceso actual quiere ejecutar la sección crítica. En caso de que el semáforo se encuentre bloqueado, se detuvo la ejecución del proceso. También permite indicar que hay que poner el semáforo en rojo.

En caso de que se necesite exclusión mutua, el semáforo dispondrá también de un sistema de espera (mediante colas de procesos) que garantice el acceso a la sección crítica de un único proceso a la vez.

En realidad, la implementación de un semáforo dependerá mucho del problema a resolver, a pesar de que la dinámica del funcionamiento sea siempre muy similar.

Además, podemos utilizar semáforos para gestionar problemas de sincronización donde un proceso deba activar la ejecución de otro o de exclusión mutua asegurando que sólo un proceso conseguirá acceder en la sección crítica para que el semáforo quedará bloqueado hasta a la salida.

También podemos usar semáforos para generar un sistema de turnos para los hilos de nuestras aplicaciones. La situación sería similar a las filas que se producen en el supermercado

cuando tenemos muchos clientes y estos deben ser atendidos por turno por el número de cajeros disponibles para pagar.

Podemos desarrollar nuestros propios semáforos que se ajusten exactamente a los requisitos de nuestras aplicaciones utilizando las técnicas que hemos visto hasta ahora: métodos y código sincronizados para garantizar bloqueos de acceso a la sección crítica, acceso atómico a las variables, guarded blocks etc.

Por suerte, Java dispone del paquete *java.util.concurrent* que proporciona una serie de clases que implementan estas funcionalidades de manera genérica. En concreto encontramos la clase *Semaphore* que implementa un semáforo de tipo contador.

Veamos un ejemplo de uso. Utilizaremos el semáforo para controlar la ejecución de hilos y no permitir que se ejecuten más de 4 a la vez. Si las acciones que ejecutan los hilos tuvieran interdependencias deberíamos controlarlas con las técnicas que hemos visto hasta ahora

```
import java.util.concurrent.Semaphore;

public class SemaphoreUsage implements Runnable {
	// Número de procesos disponibles
	private static final int AVAILABLE_THREADS = 4;
	// Semáforo, incluye el número de procesos en paralelo
	private static final Semaphore semafore = new Semaphore(AVAILABLE_THREADS);
	// Nombre del proceso
	private final String name;
	// el constructor recibe el nombre del proceso
	public SemaphoreUsage(String name) {
		this.name = name;
	}

	@Override
	public void run() {
		try {
			// pedimos permiso al semáforo para continuar
			semafore.acquire();

			System.out.println("Executing process: " + name);

			// código que ejecutaría nuestro proceso

			Thread.sleep(1000);

			System.out.println("End: " + name);

			// liberamos el semáforo al terminar
			semafore.release();

		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}

	public static void main(String[] args) {

		// lanzamos 20 procesos
		for (int i = 0; i < 20; i++) {
			new Thread(new SemaphoreUsage(Integer.toString(i+1))).start();
		}
	}

}
```

Podríamos diseñar semáforos para sincronizar dos tipos procesos haciendo que los procesos del primer tipo se ejecuten siempre antes que los procesos del tipo 2. Asegurando de esta forma que los procesos de tipo 2 empiezan a ejecutarse siempre una vez que los procesos de tipo 1 han terminado.

De igual manera, Java ya nos ofrece una serie de clases en el paquete *concurrent* con implementaciones genéricas que nos permiten este tipo de gestión de hilos:

- *CountDownLatch*. Una ayuda para la sincronización que permite que uno o más hilos esperen hasta que se complete la ejecución de una serie de operaciones realizadas por otro conjunto de hilos.
- *CyclicBarrier*. Un asistente para la sincronización que permite a un conjunto de hilos esperar al resto para alcanzar un punto de barrera común. Este objeto es útil en programas que utilizan un conjunto de tamaño fijo de hilos que deben sincronizarse de manera ocasional esperando a que todos los hilos alcancen el mismo estado antes de continuar. Se denomina cíclico porque puede ser reutilizado después de que los hilos en espera han sido liberados.
- *Phaser*. Es una implementación reutilizable de sincronización en barrera, similar a *CyclingBarrier* y *CountDownLatch* pero que soporta una utilización más flexible.

Como ejemplo, imaginemos dos procesos. Uno debe escribir *Hello*, (proceso *p1*) y el proceso *p2* debe escribir *world*. El orden correcto de ejecución es primero *p1* y después *p2*, para conseguir escribir *Hello world*. En caso de que el proceso *p1* se ejecute antes de que *p2*, ningún problema: escribe *Hello* y hace un *sendSignal* en el semáforo. Cuando el proceso *p2* ejecuta encontrará semáforo cerrado, por lo que no quedará bloqueado, podrá hacer un *sendWait* en el semáforo para liberarlo y escribirá *mundo*.

```
Proceso 1 (p1)                  Proceso 2 (p2)
initial(0)
...
System.Out.Print ("Hola ")      sendWait()
sendSignal()                    System.Out.Print("mundo")
```

¿Pero qué pasa si se ejecuta primero el proceso *p2*? Como encontrará semáforo a 0, se quedará bloqueado al hacer la petición llamando *sendWait* hasta que el proceso p1 escriba *Hola* y haga el *sendSignal*, desbloqueando el semáforo. Entonces p2, que estaba bloqueado, se activará, pondrá el semáforo de nuevo a rojo y escribirá *mundo* en la pantalla.

Podríamos programar un semáforo específico para este problema, pero vemos que la clase *CountDownLatch* se ajusta a nuestro caso, veamos cómo utilizarla para resolverlo:

```
import java.util.concurrent.CountDownLatch;

public class CountDownUsage implements Runnable {
	// mensaje a escribir por el proceso
	private String msg;
	// countDownLatch para controlar la ejecución de cada fase
	private CountDownLatch countDown;

	// constructor
	public CountDownUsage(String msg, CountDownLatch countDown) {
		this.msg = msg;
		this.countDown = countDown;
	}

	public static void main(String[] args) {
		System.out.println("*** the beginning ***");
		// creamos nuestro pestillo que se activará tras
		// la ejecución de un hilo en la fase 1
		// "hello"
		final CountDownLatch countDownPhase1 = new CountDownLatch(1);
		// creamos nuestro pestillo que se activará tras
		// la ejecución de un hilo en la fase 2
		// "world"
		final CountDownLatch countDownPhase2 = new CountDownLatch(1);

		// declaramos y lanzamos el proceso P1: hello
		new Thread(new CountDownUsage("Hello ", countDownPhase1)).start();
		// declaramos el proceso P2: world
		 Thread t2 = new Thread(new CountDownUsage(" world!", countDownPhase2));

		try {
			// esperamos a la finalización de la fase 1
			// antes de lanzar la segunda
			countDownPhase1.await();
			t2.start();
			countDownPhase2.await();
			System.out.print("\n");
			System.out.println("*** the end ***");

		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}

	@Override
	public void run() {
		try {
			Thread.sleep(1000);
			System.out.print(msg);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		countDown.countDown();
	}

}
```

En una implementación más real es muy probable que las clases para los procesos de tipo 1 y tipo 2 fueran diferentes y que la propia ejecución de los hilos de tipo 2 en el método run validaran el estado de la cuenta atrás de la fase 1.

7.9.2 Monitores

Otra forma de resolver la sincronización de procesos es el uso de monitores. Los monitores son un conjunto de procedimientos encapsulados que nos proporcionan el acceso a recursos compartidos a través de varios procesos en exclusión mutua.

Las operaciones del monitor están encapsuladas dentro de un módulo para protegerlas del programador. Únicamente un proceso puede estar en ejecución dentro de este módulo.

El grado de seguridad es alto ya que los procesos no saben cómo están implementados estos módulos. No se necesitará saber de antemano cómo y en qué momento se llamará a las operaciones del módulo, así es más robusto. Un monitor, una vez implementado, si está diseñado correctamente siempre funcionará bien.

Un monitor se puede ver como una habitación, cerrada con una puerta, que tiene dentro de los recursos. Los procesos que quieran utilizar estos recursos deben entrar en la habitación, pero con las condiciones que marca el monitor y únicamente un proceso a la vez. El resto que quiera hacer uso de los recursos deberá espera que salga lo que está dentro.

Por su encapsulación, la única acción que debe tomar el programador del proceso que quiera acceder al recurso protegido es informar al monitor. La exclusión mutua está implícita. Los semáforos, en cambio, se deben implementar con una secuencia correcta de señal y esperar el fin de no bloquear el sistema.

Un **monitor** es un algoritmo que hace una abstracción de datos que nos permite representar de forma abstracta un recurso compartido mediante un conjunto de variables que definen su estado. El acceso a estas variables únicamente es posible desde unos métodos del monitor.

Los monitores deben incorporar un mecanismo de sincronización, por ejemplo, haciendo uso del uso de señales. Estas señales se utilizan para impedir los bloqueos. Si el proceso que es en el monitor debe esperar una señal, se pone en estado de espera o bloqueado fuera del monitor, permitiendo que otro proceso haga uso del monitor. Los procesos que están fuera del monitor, están a la espera de una condición o señal para volver a entrar.

Estas variables que se utilizan para las señales y son utilizadas por el monitor para la sincronización se llaman **variables de condición**. Estas pueden ser manipuladas con operaciones de *sendSignal* y *sendWait* (como los semáforos).

- *sendWait*: un proceso que está esperando a un evento indicado por una variable de condición abandona de forma temporal el monitor y se pone a la cola que corresponde a su variable de condición.
- *sendSignal*: desbloquea un proceso de la cola de procesos bloqueados con la variable de condición indicada y se pone en estado preparado para entrar en el monitor. El proceso que entra no debe ser el que más tiempo lleva esperando, pero se debe garantizar que el tiempo de espera de un proceso sea limitado. Si no existe ningún proceso a la cola, la operación *sendSignal* no tiene efecto, y el primer proceso que pida el uso del monitor entrará.

Un monitor consta de 4 elementos:

- Variables y métodos internos al monitor que sólo son accesibles desde dentro del monitor. No se modifican entre dos llamadas consecutivas al monitor.

- Código de inicialización: inicializa las variables permanentes, se ejecuta cuando el monitor es creado.
- Métodos externos o exportados: son métodos que son accesibles desde fuera del monitor por los procesos que quieren entrar a hacer uso.
- Cola de procesos: es la cola de procesos bloqueados a la espera de la señal que los libere para volver a entrar en el monitor.

En Java podemos implementar fácilmente monitores con las técnicas que ya hemos visto. Por ejemplo, una clase en los que todos sus métodos públicos son *synchronized* ya sigue el patrón de un monitor garantizando la exclusión mutua. Los métodos *wait* y *notifyAll*, permiten enviar las señales de sincronización del monitor.

Un semáforo es un objeto que permite sincronizar el acceso a un recurso compartido y un monitor es una interfaz de acceso al recurso compartido. Son el encapsulamiento de un objeto, por lo que hace un objeto más seguro, robusto y escalable.

8 El Modelo Productor-Consumidor

En computación, el **problema del productor-consumidor** es un ejemplo clásico de problema de sincronización de multiprocesos. El programa describe dos procesos, productor y consumidor, ambos comparten un buffer de tamaño finito. La tarea del productor es generar un producto, almacenarlo y comenzar nuevamente; mientras que el consumidor toma (simultáneamente) productos uno a uno. El problema consiste en que el productor no añada más productos que la capacidad del buffer y que el consumidor no intente tomar un producto si el buffer está vacío.

La idea para la solución es la siguiente, ambos procesos (productor y consumidor) se ejecutan simultáneamente y se “despiertan” o “duermen” según el estado del buffer. Concretamente, el productor agrega productos mientras quede espacio y en el momento en que se llene el buffer se pone a “dormir”. Cuando el consumidor toma un producto notifica al productor que puede comenzar a trabajar nuevamente. En caso contrario si el buffer se vacía, el consumidor se pone a dormir y en el momento en que el productor agrega un producto crea una señal para despertarlo. Se puede encontrar una solución usando mecanismos de comunicación interprocesos, generalmente se usan semáforos. Una inadecuada implementación del problema puede terminar en un deadlock donde ambos procesos queden en espera de ser despertados. Este problema pude ser generalizado para múltiples consumidores y productores.

Veamos un ejemplo de implementación con un único mensaje en el buffer. En este caso el producto o dato es una serie de mensajes de texto, que se comparte a través de un objeto de tipo *Drop*.

```
public class Drop {
    // End Message
    public static final String DONE = ".";
    // Message sent from producer
    // to consumer.
    private String message;
    // True if consumer should wait
    // for producer to send message,
    // false if producer should wait for
    // consumer to retrieve message.
    private boolean empty = true;

    public synchronized String take() {
        // Wait until message is
        // available.
        while (empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = true;
        // Notify producer that
        // status has changed.
        notifyAll();
        return message;
    }

    public synchronized void put(String message) {
        // Wait until message has
        // been retrieved.
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = false;
        // Store message.
        this.message = message;
        // Notify consumer that status
        // has changed.
        notifyAll();
    }
}
```

El hilo productor, definido por *Producer* envía una serie de mensajes. La cadena "." indica que todos los mensajes han sido enviados. Para simular situaciones impredecibles como las que podrían ocurrir en el mundo real, el hilo se detiene por un intervalo aleatorio de tiempo entre mensajes.

```
import java.util.Random;

public class Producer implements Runnable {

	private Drop drop;

	public Producer(Drop drop) {
		this.drop = drop;
	}

	@Override
	public void run() {
		String importantInfo[] = { "Programas", "Procesos"
			, "Servicios", "Hilos" };
		Random random = new Random();

		for (int i = 0; i < importantInfo.length; i++) {
			drop.put(importantInfo[i]);
			try {
				Thread.sleep(random.nextInt(5000));
			} catch (InterruptedException e) {
			}
		}
		drop.put(Drop.DONE);
	}
}
```

El hilo consumidor simplemente recibe los mensajes y los imprime en la salida del programa (consola) hasta que reciba el mensaje ".". Este hilo también se para durante periodos aleatorios de tiempo.

```
public class Consumer implements Runnable {
    private Drop drop;

    public Consumer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        Random random = new Random();
        for (String message = drop.take();
             ! message.equals(Drop.DONE);
             message = drop.take()) {
            System.out.format("MESSAGE RECEIVED: %s%n", message);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
    }
}
```

Finalmente tenemos el hilo principal, definido por *ProducerConsumerExample*, que lanza los hilos productor y consumidor.

```
public class ProducerConsumerExample {

	public static void main(String[] args) {
		Drop drop = new Drop();
	  (new Thread(new Producer(drop))).start();
	  (new Thread(new Consumer(drop))).start();
	}

}
```

9 Concurrencia de alto nivel en Java

9.1 Locks

El paquete *java.util.concurrent.locks* contiene una serie de interfaces y clases que proporcionan un framework de bloqueos y condiciones de espera que ofrecen mecanismos de sincronización de hilos más flexibles y sofisticados que los bloqueos implícitos que proporcionan el código sincronizado (*synchronized*).

Veamos algunos aspectos de la interfaz más básica: *Lock*.

Lock funciona de manera muy similar a los bloqueos implícitos proporcionados por *sychronized*; sólo un hilo puede tener un objeto *Lock* al mismo tiempo. También soportan un mecanismo de *wait/notify* utilizando los objetos asociados *Condition*.

La mayor ventaja de los objetos *Lock* es la posibilidad de echarse atrás de un intento de adquirir un bloqueo. El método *tryLock* se retira si el bloqueo no está disponible inmediatamente (o antes de que pase un determinado tiempo si especificamos un timeout). El método *lockInterruptibly* retira su petición de bloqueo si otro hilo envía una interrupción antes de que el bloqueo se haya adquirido.

La clase *ReentrantLock* es la principal implementación de la interfaz *Lock*. Veamos cómo podemos usar esta clase para la sincronización de hilos:

```
public class SharedObject {
    //...
    ReentrantLock lock = new ReentrantLock();
    int counter = 0;

    public void perform() {
        lock.lock();
        try {
            // Sección crítica
            count++;
        } finally {
            lock.unlock();
        }
    }
    //...
}
```

Tenemos que asegurar que ejecutamos un *unlock()* después de bloquear el candado (*lock()*), para ello utilizaremos un bloque *try-finally*.

Veamos un ejemplo de uso de estos objetos para resolver el problema de Deadlock que vimos con los saludos entre los amigos en el apartado Deadlocks.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class DeadlockSafe {

	static class Friend {
		private final String name;
		private Lock lock;

		public Friend(String name) {
			this.name = name;
			this.lock = new ReentrantLock();
		}

		public String getName() {
			return this.name;
		}

		public boolean tryBow(Friend bower) {
			boolean myLock = false;
			boolean bowerLock = false;
			// tratamos de conseguir el bloqueo de los Lock
			// de ambos objetos
			try {
				myLock = lock.tryLock();
				bowerLock = bower.lock.tryLock();
			} finally {
				// Si no conseguimos el bloqueo de ambos
				// desbloqueamos aquel Lock
				// que hubieramos bloqueado
				if (!(myLock && bowerLock)) {
					if (myLock) {
						lock.unlock();
					}
					if (bowerLock) {
						bower.lock.unlock();
					}
				}
			}
			return myLock && bowerLock;
		}

		public void bow(Friend bower) {
			System.out.format("%s: %s" + " has bowed to me!%n"
				, this.name, bower.getName());
			bower.bowBack(this);
		}

		public void bowBack(Friend bower) {
			System.out.format("%s: %s" + " has bowed back to me!%n"
				, this.name, bower.getName());
		}
	}
```

```
	// declaramos una clase runnable para lanzar los saludos
	static class Bow implements Runnable {
		// declaramos dos objetos Friend: yo y mi amigo
		public Friend me;
		public Friend myFriend;

		public Bow(Friend me, Friend myFriend) {
			this.me = me;
			this.myFriend = myFriend;
		}

		@Override
		public void run() {
			// tratamos de obtener ambos bloqueos
			boolean locked = me.tryBow(myFriend);
			if (locked) {
				// Si lo conseguimos, saludamos y desbloqueamos
				try {

					me.bow(myFriend);
				} finally {
					// el bloque finally asegura el desbloqueo
					me.lock.unlock();
					myFriend.lock.unlock();
				}
			} else {
				// si tratamos de saludar pero alguno
				// y ya había bloqueo
				// detectamos la situación
				// y devolvemos un mensaje
				System.out.println(me.name
					+ ": I was trying to bow my friend "
					+ myFriend.name
					+ " at the moment he was bowing me");
			}

		}

	}

	public static void main(String[] args) {
		final Friend pedro = new Friend("Pedro");
		final Friend pablo = new Friend("Pablo");

		// arrancamos el saludo de pedro a plablo
		new Thread(new Bow(pedro, pablo)).start();
		// arrancamos el saludo de pedro a plablo
		new Thread(new Bow(pablo, pedro)).start();
	}
}
```

9.2 Executors

Una manera de resolver un problema de manera eficiente es dividirlo en problemas más pequeños y cada uno de estos subproblemas enviarlos a ejecutar por separado. Así, de una tarea complicada resultan varias de más simples. Para poder ejecutar en paralelo todas estas subtareas y así aumentar el rendimiento, se crean diferentes subprocesos o hilos, pero esta creación es totalmente transparente para el programador, es la máquina virtual de Java la encargada de gestionar la creación de hilos para ejecutar en paralelo las subtareas.

Al ejecutar estos procesos de forma paralela en los distintos núcleos de nuestro ordenador podremos mejorar el rendimiento de nuestra aplicación.

Hemos visto cómo Java proporciona apoyo a la programación concurrente con librerías de bajo nivel a través de la clase *java.lang.Thread* y de la interfaz *java.lang.Runnable*. El uso de *Thread* y *Runnable* conlleva un esfuerzo adicional del programador, obligándole a añadir pruebas extras para verificar el comportamiento correcto del código, evitando lecturas y escrituras erróneas, sincronizando las ejecuciones de los procesos y evitando los problemas derivados de los bloqueos.

Esto funciona bien para pequeñas aplicaciones, pero para aplicaciones de gran escala, tiene sentido separar la creación y gestión de los hilos del resto de la aplicación. Los objetos que encapsulan estas funciones se conocen como *Executors*.

9.2.1 Interfaces Executor

El paquete *java.util.concurrent* define tres interfaces *executor*:

- *Executor*, una interfaz simple que suporta el lanzamiento de nuevas tareas
- *ExecutorService*, una subinterfaz de *Executor* que añade funcionalidades que ayudan a gestionar el ciclo de vida de las tareas y del executor.
- *ScheduledExecutorService*, una subinterfaz de *ExecutorService* que soporta ejecuciones futuras y/o periódicas de las tareas.

Interfaz Executor

La interfaz *Executor* ofrece un único método, *execute*, diseñado para reemplazar a la creación de hilos. Si *r* es un objeto *Runnable*, y *e* es un objeto *Executor* se puede reemplazar:

```
(new Thread(r)).start();
```

por

```
e.execute(r);
```

Sin embargo, la definición de *execute* es menos específica. La estrategia de bajo nivel crea un nuevo hilo y lo lanza inmediatamente. Dependiendo de la implementación del *Executor*, *execute* puede hacer la misma cosa, pero es más deseable usar un hilo trabajador existente para hacer correr *r*, o para poner a *r* en la cola a la espera de un hilo trabajador disponible (Veremos el concepto de hilo trabajador a continuación en el apartado Thread pools).

Las implementaciones de *executor* en *java.util.concurrent* están diseñadas para hacer completo uso de las interfaces más avanzadas *ExecutorService* y *ScheduledExecutorService*, aunque éstas también trabajan con la interfaz base *Executor*.

Interfaz ExecutorService

La interfaz *ExcecutorService* complementa a *execute* el método *submit*. Igual que *execute*, *submit* acepta objetos *Runnable*, pero también acepta objetos *Callable*, los cuales permiten que las tareas retornen valores. El método *submit* retorna un objeto *Future*, que es usado para recibir el valor de retorno del objeto *Callable* y permite gestionar el estado de ambas tareas *Callable* y *Runnable*.

ExecutorService también ofrece métodos para enviar grandes colecciones de objetos *Callable*. Finalmente, *ExecutorService* ofrece un número de métodos para administrar el apagado del executor. Para soportar apagado inmediato, las tareas deberían gestionar las interrupciones correctamente.

Interfaz ScheduledExecutorService

La interfaz *ScheduledExecutorService* complementa los métodos de su interfaz padre *ExecutorService* con *shedule*, el cual ejecuta una tarea *Runnable* o *Callable* después de un retraso especificado. Además, define los métodos *scheduleAtFixedRate* y *scheduleWithFixedDelay*, los cuales ejecutan tareas especificadas repetidamente a intervalos definidos.

9.2.2 Thread pools

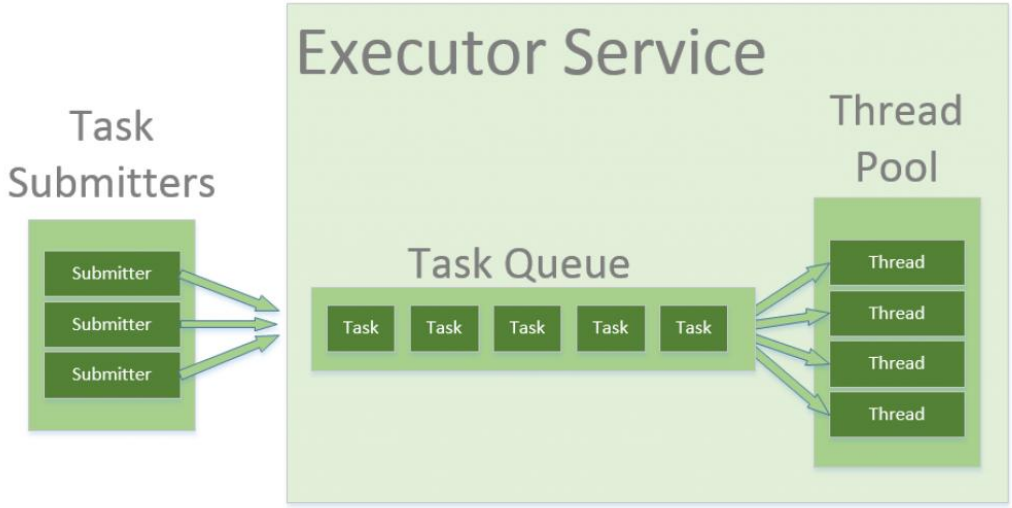
La mayoría de las implementaciones de la interfaz executor utilizan *thread pools* que constan de *worker thread* o hilos trabajadores. La utilización de worker thread minimiza el coste computacional de la creación de hilos. Los hilos utilizan una cantidad significativa de memoria y en una aplicación grande asignar y desasignar objetos de tipo *Thread* ocasiona un coste en la gestión de la memoria considerable.

Vamos a ver tres tipos de gestores:

- uno genérico, *ThreadPoolExecutor*
- uno capaz de seguir pautas temporales de ejecución, *ScheduledThreadPoolExecutor*
- y uno capaz de ejecutar programación paralela concurrente, *ForkJoinPool*.

ThreadPoolExecutor, es la clase base que permite implementar el gestor genérico. Los gestores genéricos son adecuados para solucionar problemas que se puedan resolver ejecutando tareas independientes entre sí. Imaginemos que queremos saber la suma de una lista grande de números. Podemos optar por sumar secuencialmente la lista o bien para dividirla en dos o tres trozos y sumar cada trozo por separado. Al final sólo habrá que añadir el valor de cada trozo al resultado. La suma de cada trozo es totalmente independiente y por lo tanto puede realizarse con independencia del resto. Es decir, sólo habría que crear tantas tareas como trozos a sumar y pasarlas al gestor para que las ejecute.

ThreadPoolExecutor, no crea un hilo para cada tarea (instancia de *Callable* o *Runnable*). De hecho, es habitual asignarle más tareas que no hilos. Las tareas pendientes permanecen en una cola de tareas y un número de hilos determinado encarga de irlas ejecutando. La decisión de cuántos hilos crear será una decisión difícil y dependerá en cada caso, considerando el número de procesadores disponibles.



La clase *Executors* aglutina un conjunto de utilidades en forma de métodos estáticos y permite tres formas rápidas de configurar las instancias de *ThreadPoolExecutor*:

- *newCachedThreadPool*(). Crea un *pool* que va creando hilos a medida que los necesita y reutiliza los hilos inactivos.
- *newFixedThreadPool*(int numThreads). Crea un *pool* con un número de hilos fijo. El indicado en el parámetro. Las tareas se van agregando a una cola de tareas y se van asociando los hilos a medida que estos quedan libres.
- *newSingleThreadExecutor*(). Crea un *pool* con un solo hilo que será el que procese todas las tareas.

Cualquiera de estas configuraciones obtendrá una instancia de *ThreadPoolExecutor* que se comportará de la manera descrita.

Algunos de los métodos de la clase *ThreadPoolExecutor* son:

- ***execute***. Ejecuta la tarea dada en algún momento en el futuro.
- ***invokeAll***. Ejecuta las tareas dadas, devolviendo una lista de futuros que mantienen su estado y resultados cuando todo se completa o el tiempo de espera expira, lo que ocurra primero. Future.isDone() es verdadero para cada elemento de la lista devuelta. Al regresar, las tareas que no se han completado se cancelan. Tenga en cuenta que una tarea *completada* podría haber finalizado normalmente o lanzando una excepción. Los resultados de este método no están definidos si la colección dada se modifica mientras esta operación está en progreso.
- ***getPoolSize***. Devuelve el número actual de subprocesos en el grupo.
- ***getCompletedTaskCount()***. Devuelve el número de tareas que han completado la ejecución.

ScheduledThreadPoolExecutor es un caso específico de *pool*. Se trata de un gestor de hilos, con una política de ejecución asociada a una secuencia temporal. Es decir, ejecuta las tareas programadas cada cierto tiempo. Puede ser de una sola vez o de forma repetitiva. Por ejemplo, si queremos consultar nuestro correo cada 5 minutos, mientras realizamos otras tareas, podemos programar un hilo independiente con esta tarea

usando *ScheduledThreadPoolExecutor*. La tarea podría comprobar el correo y avisar sólo en caso de que hayan llegado nuevos. De esta manera podemos concentrarnos en nuestro trabajo dejando que el hilo periódico trabaje por nosotros.

Podemos obtener instancias de *ScheduledThreadPoolExecutor* usando el método estático *newScheduledThreadPool* de la clase *Executors*.

La clase *ScheduledThreadPoolExecutor* dispone de los siguientes métodos para gestionar el retraso y cadencia:

- ***schedule (Callable task, long delay, TimeUnit timeunit).*** Crea y ejecuta una tarea (*task*) que es invocada tras un tiempo concreto (*delay*) expresado en las unidades indicadas (*timeunit*).
- ***schedule (Runnable task, long delay, TimeUnit timeunit).*** Crea y ejecuta una acción (*task*), una sola vez, después de que haya transcurrido un tiempo concreto (*delay*) expresado en las unidades indicadas (*timeunit*).
- ***scheduleAtFixedRate (Runnable task, long initialDelay, long period, TimeUnit timeunit).*** Crea y ejecuta una tarea (*task*) de forma periódica. La primera vez se invocará tras un retraso inicial (*initialDelay*) y posteriormente cada período de tiempo determinado (*period*). El tiempo se mide en las unidades indicadas (*timeunit*).
- ***scheduleWithFixedDelay (Runnable task, long initialDelay, long delay, TimeUnit timeunit)***. Crea y ejecuta una tarea (*task*) de forma periódica. La primera vez se invocará tras un retraso inicial (*initialDelay*) y posteriormente, al terminar la última tarea lanzada no se empezará la siguiente hasta que no haya pasado el tiempo indicado (*delay)*. El tiempo se mide en las unidades indicadas (*timeunit*).

Los métodos que reciben tareas de tipo Callable devolverán el resultado obtenido durante el cálculo.

Ejemplo con *ThreadPoolExecutor*

Vamos ahora a ver un ejemplo para calcular diez multiplicaciones aleatorias de enteros usando un *Executor* con un máximo de 3 hilos.

Podemos instanciar la clase threadpool utilizado la clase Excutors:

```
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(3);
```

Esto significa que, si hay más de 3 tareas para ejecutar, únicamente enviará 3 a ejecutar y el resto se quedarán bloqueadas hasta que un hilo termine su procesamiento.

La clase interior *Multiplication* implementa la interfaz *Callable*. Se podría utilizar el método *run* e implementar *Runable*. La diferencia es que *Callable* devuelve valores del resultado en forma de cualquier objeto. La concurrencia se realiza dentro del método *call*.

Se crea una lista de diez multiplicaciones al azar. Y luego llama al método de ejecutor *invokeAll*. Este recibe una colección de tareas de tipo *Callable*, las ejecuta y devuelve el resultado dentro de un objeto *Future*. *Future* es una interfaz diseñada para apoyar a las clases que mantengan datos calculados de forma asíncrona. Las instancias de *Future* obligan a utilizar los métodos *get* para acceder a los resultados. Esto asegura que el valor real sólo será

accesible cuando los cálculos hayan terminado. Mientras duren los cálculos la invocación del método *get* implicará un bloqueo del hilo que la esté invocando.

Los objetos *Future* se generan como consecuencia de ejecutar el método *call* de las instancias *Callable*. Por cada *Callable* invocada se genera un objeto *Future* que permanecerá bloqueado hasta que la instancia *callable* asociada finalice.

ThreadPoolExecutor dispone también del método *invokeAny(tasks)* que recibe una colección de instancias de *Callable* que irá ejecutando hasta que una de ellas finalice sin error. En este momento devolverá el resultado obtenido por la tarea finalizada en un objeto *Future*.

El método *shutdown* inicia un cierre ordenado en el que se ejecutan tareas enviadas previamente, pero no se aceptarán nuevas tareas.

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

public class MultiplyList {

	// La clase Multiplication implementa la interfaz Callable<Integer>
	// por lo tanto devolverá un entero la ejecución del método call
	// la tarea realizará la multiplicación de dos enteros
	static class Multiplication implements Callable<Integer> {
		private int op1;
		private int op2;

		public Multiplication(int operador1, int operador2) {
			this.op1 = operador1;
			this.op2 = operador2;
		}

		// Implementación de la interfaz
		@Override
		public Integer call() throws Exception {
			return op1 * op2;
		}
	}

	public static void main(String[] args) {
		//Creamos la instacia de ThreadPoolExecutor con 3 hilos
		//Este número dependerá de nuestra necesidad de paralelización
		//y de los nucleos disponibles en la máquina donde se va a ejecutar
		ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(3);
		//Creamos un array list para las tareas que queremos ejecutar de manera paralela
		//objetos callable
		List<Multiplication> taskList = new ArrayList<Multiplication>();
		//Creamos las tareas a ejecutar; 10 mulitplicaciones de 2 números aleatorios
		for (int i = 0; i < 10; i++) {
			int op1 = (int) (Math.random() * 10);
			int op2 = (int) (Math.random() * 10);
			Multiplication calcula = new Multiplication(op1, op2);
			System.out.println("Task " + i + " is:" + op1 + " * " + op2);
			taskList.add(calcula);
		}
		//Creamos la lista para los resultados, contendrá objetos de tipo Future
		List<Future<Integer>> resultList;
		try {
			//Lanzamos todas las tareas para su ejecución
			//la ejecución de cada tarea es independiente de las demás
			resultList = executor.invokeAll(taskList);
			//Pedimos al ejecutor que finalize
			executor.shutdown();
			//Mostramos los resultados
			for (int i = 0; i < resultList.size(); i++) {
				Future<Integer> res = resultList.get(i);
				System.out.println("Result of task " + i + " is:" + res.get());
			}
		} catch (InterruptedException e) {
			e.printStackTrace();
		} catch (ExecutionException e1) {
			e1.printStackTrace();
		}
	}

}
```

Ejemplo con ScheduledThreadPoolExecutor

En el siguiente ejemplo utilizaremos el ejecutor *ScheduledThreadPoolExecutor* para programar una tarea que se ejecuta periódicamente, tras un retraso programado.

Primero creamos un *pool* de un único hilo del *ScheduledThreadPoolExecutor* y utilizamos *scheduleAtFixedRate*, que nos permite programar y ejecutar el inicio de la tarea (initialDelay) y programar las tareas sucesivas después de la primera finalización (*long period*).

Ejecuta el método *run* de *WatcherTask*, aa primera vez al cabo de 1 segundos y posteriormente cada 5 segundos.

Por último, se llama el método *shutdown* para terminar la ejecución.

```
import java.io.File;
import java.util.Date;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;

public class FileWatcher {
	// clase para la tarea
	static class WatcherTask implements Runnable {
		// path de la carpeta que va a controlar
		private String path = "";

		// constructor con el parámetro
		public WatcherTask(String folder) {
			super();
			this.path = folder;
		}

		@Override
		public void run() {
			// mostramos el momento de la ejecución
			System.out.println("Watcher executed at : " + new Date());
			System.out.println("<<<<<<<<<<");
			// objeto file inicializado en
			// la ruta definida
			File f = new File(path);
			// si existe la ruta y corresponde con un directorios
			if (f.exists() && f.isDirectory()) {
				// devolvemos la lista de elementos que contiene
				String[] files = f.list();
				// mostramos los ficheros presentes
				for (int i = 0; i < files.length; i++) {
					System.out.println(files[i]);
				}
			} else {
				System.out.println("Directory does not exists: " + path);
			}
			System.out.println(">>>>>>>>>>");
		}
	}
```

```
	public static void main(String[] args) {
		// creamos un ScheduledThreadPool de un hilo
		ScheduledExecutorService executor =
			Executors.newScheduledThreadPool(1);
		// creamos la tarea para el watcher
		WatcherTask task = new WatcherTask("/home/administrador/Documents/");
		// Mostramos la hora de ejecución
		System.out.println("The time is : " + new Date());
		// lanzamos la ejecución
		ScheduledFuture<?> result =
			executor.scheduleAtFixedRate(task, 1, 5, TimeUnit.SECONDS);

		// esperamos un minuto antes de terminar
		try {
			Thread.sleep(60000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}

		// terminamos la ejecución del thread pool
		executor.shutdown();
	}

}
```

9.2.3 Fork-Join

El framework Fork-Join es una evolución de los patrones *Executor*. La clase *ForkJoinPool* es el gestor de hilos central de este framework. Procesa las tareas con el algoritmo por robo (*work-Stealing*). Esto significa que el gestor del *pool* busca hilos poco activos e intercambia tareas en espera. Además, las tareas pueden crear nuevas tareas que se incorporarán a la cola de tareas pendientes para ser ejecutadas. La eficiencia conseguida es bastante alta, ya que se aprovecha al máximo el procesamiento paralelo. Por este motivo Fork-Join es una alternativa ideal para algoritmos que puedan resolverse de forma recursiva ya que se conseguirá la máxima eficiencia intercambiando aquellas tareas que se encuentren esperando resultados por otros que permitan avanzar en los cálculos.

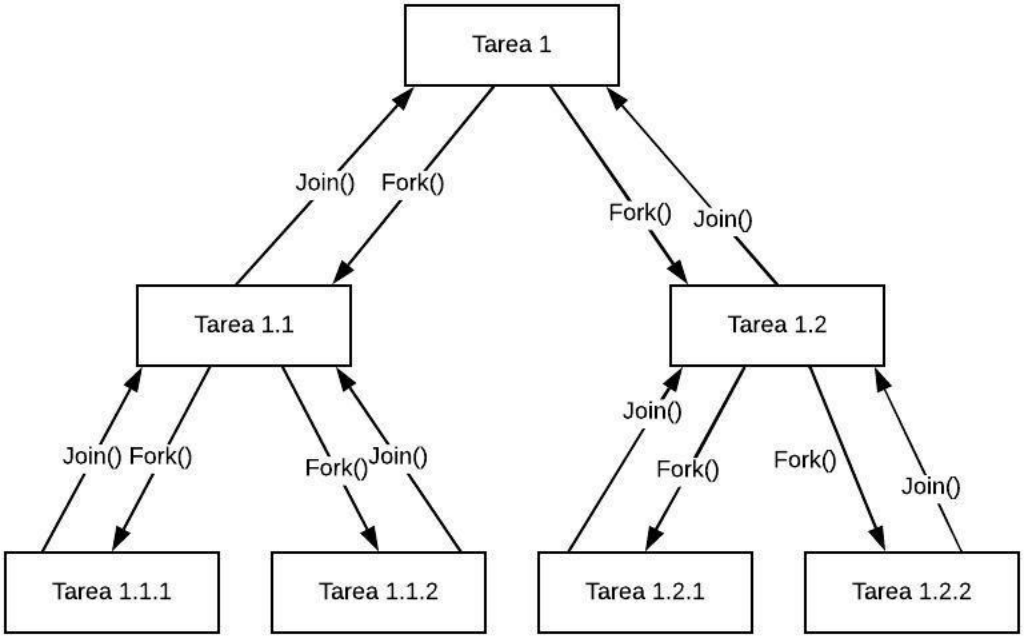
La clase *ForkJoinTask* es una clase abstracta para las tareas que se ejecutan en *ForkJoinPool* y contiene los métodos *fork* y *join*. El método *fork()* sitúa la tarea invocada a la cola de ejecuciones en cualquier momento para que sea planificada. Esto permite que una tarea crear otras nuevas y dejarlas listas para ser ejecutadas cuando el gestor lo considere.

El método *join()* detendrá la ejecución del hilo invocador a la espera que la tarea invocada finalice la ejecución y devuelva en su caso los resultados. El bloqueo del hilo pondrá en alerta al gestor *ForkJoinPool* que podrá intercambiar la tarea parada por otra que quede en espera.

ForkJoinTask está implementada por dos clases que el especializan, la clase *RecursiveTask* y la clase *RecursiveAction*. La primera es adecuada para tareas que necesiten calcular y devolver un valor. La segunda en cambio representa procedimientos o acciones que no necesitan devolver ningún resultado.

Cuando usamos *RecursiveAction* resulta muy práctico utilizar el método *invokeAll*, en vez de llamar, por cada tarea generada los métodos *fork* y *join*. Concretamente, el método *invokeAll* hace una invocación a *fork* por cada una de las tareas recibidas como parámetro y seguidamente se espera a la finalización de cada hilo con una llamada al método *join* de las

tareas emprendidas. El método *invokeAll* no devuelve ningún resultado, por lo tanto, no resulta muy útil utilizar con objetos de tipo *RecursiveTask*.



Uso básico

El primer paso para usar el framework *fork/join* es escribir código que ejecuta un bloque de trabajo. Deberíamos escribir código similar al siguiente pseudocódigo:

```
if (mi parte del trabajo es suficientemente pequeña)
      hacer el trabajo directamente
else
      dividir mi trabajo en dos partes
      invocar a las dos partes y esperar los resultados
```

Veamos un ejemplo de uso. Buscaremos dentro de un array de enteros el número mayor. La recursividad se basará en dividir la colección de enteros en dos y pedir por cada trozo la obtención del valor máximo y el cálculo secuencial en un algoritmo de búsqueda iterativo.

Usaremos *RecursiveTask* porque necesitamos devolver el valor máximo. Los valores de partida los pasaremos siempre a través del constructor de la tarea y los almacenaremos como atributos para tenerlos disponibles en el método *compute* que es el equivalente específico de las tareas *ForkJoinTask* a los métodos *run* o *call* ya vistos para *Executor*.

```
import java.util.Random;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class MaxNumberTask extends RecursiveTask<Integer> {
	// constantes para el tamaño total del array
	// y el límite para la búsqueda iterativa
	final static int TOTAL = 100_000_000;
	final static int THRESHOLD = 10_000_000;

	// declaramos el array de números
	int[] numbers;
	// el valor inicial a evaluar
	int first;
	// el número de resgistros a evaluar;
	int length;

	// constructor
	public MaxNumberTask(int[] numbers, int first, int length) {
		this.numbers = numbers;
		this.first = first;
		this.length = length;
	}

	// obtenemos el máximo de manera iterativa
	private int getMaxIterative() {

		int max = Integer.MIN_VALUE;
		// recorremos el número de elementos indicado (length)
		// desde el primero (first)
		int high = first + length;
		for (int i = first; i < high; i++) {
			if (numbers[i] > max) {
				max = numbers[i];
			}
		}
		return max;
	}

	// obtenemos el máximo de manera recursiva
	private int getMaxRecursive() {
		// dividimos el intervalo y lanzamos una tarea para cada mitad
		int mid = length / 2;
		MaxNumberTask task1 = new MaxNumberTask(numbers, first, mid);
		task1.fork();
		MaxNumberTask task2 =
			new MaxNumberTask(numbers, first + mid, length - mid);
		task2.fork();
		// el resultado será el máximo de los valores
		// máximos de cada intervalo
		return Math.max(task1.join(), task2.join());
	}

	static int[] generateArray(int size) {
		int[] numbers = new int[size];
		// preparamos el objeto Ramdom
		Random r = new Random(System.currentTimeMillis());
		// rellenamos con números aleatorios
		for (int i = 0; i < numbers.length; i++) {
			numbers[i] = r.nextInt();
		}
		return numbers;

	}
```

```
        @Override
        protected Integer compute() {
                // si estamos por arriba del umbral
                // aplicamos recursividad
                // si no recorremos el array y devolvemos el mayor
                Integer max = 0;
                if (length > THRESHOLD) {
                        max = getMaxRecursive();
                } else {
                        max = getMaxIterative();
                }
                return max;
        }

        public static void main(String[] args) {
                int[] numbers = generateArray(TOTAL);

                // Creamos la tarea a realizar
                MaxNumberTask task = new MaxNumberTask(numbers, 0, TOTAL);

                // búsqueda secuencial
                long start = System.currentTimeMillis();
                int max = task.getMaxIterative();
                long end = System.currentTimeMillis();
                System.out.println("Max number (" + max
                        + ") iterative found in " + (end - start));

                // busqueda recursiva en paralelo con fork - join
                ForkJoinPool pool = new ForkJoinPool();
                start = System.currentTimeMillis();
                Integer res = pool.invoke(task);
                end = System.currentTimeMillis();
                System.out.println("Max number (" + res
                        + ") recursive parallel found in " + (end - start));

        }

}
```

9.3 Colecciones concurrentes

El paquete *java.util.concurrent* incluye un número adicional de colecciones al framework Java Collections. Son las siguientes:

- *BlockingQueue* define una estructura de datos First-in-First-out (FIFO) que bloquea o pausa cuando se intenta agregar un elemento a una cola llena o retirar uno de una cola vacía.
- *ConcurrentMap* es una subinterfaz de *java.util.Map* que define operaciones atómicas. Estas operaciones eliminan o reemplazan un par clave-valor sólo si la clave está presente, o agrega un par clave-valor sólo si la clave está ausente. Hacer estas operaciones atómicas ayuda a evitar sincronización. La implementación estándar de uso general de *ConcurrentMap* es *ConcurrentHashMap*, que es la colección análoga concurrente de *HashMap*.
- *ConcurrentNavigableMap* es una subinterfaz de *ConcurrentMap* que soporta coincidencias aproximadas. La implementación estándar de propósito general de *ConcurrentNavigableMap* es *ConcurrentSkipListMap*, que es la colección análoga concurrente de *TreeMap*.

Todas estas colecciones ayudan a evitar errores de consistencia de memoria definiendo relaciones happens-before entre una operación que añade un objeto a la colección con las operaciones posteriores de acceso o borrado.

9.4 Variables atómicas

El paquete *java.util.concurrent.atomic* define clases que soportan operaciones atómicas sobre variables individuales. Todas las clases tienen métodos *get* y *set* que funcionan igual que la lectura y escritura de variables *volatile*. Esto quiere decir que un *set* tiene una relación happens-before con cualquier *get* posterior sobre la misma variable. El método atómico *compareAndSet* también tiene estas características de consistencia de memoria, como lo hacen los métodos aritméticos atómicos simples que se aplican a variables atómicas enteras.

Para ver cómo debería ser usado este paquete, vamos a volver sobre la clase *Counter* que originalmente usamos para demostrar la interface hilo:

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

Una forma para hacer *Counter* seguro para para el uso desde hilos es hacer sus métodos sincronizados tal como *SynchronizedCounter*:

```
class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }

}
```

Para esta clase simple, la sincronización es una solución aceptable. Pero para clases más complicadas, podríamos querer evitar el impacto de bloqueos e inanición de la sincronización innecesaria. Reemplazar el tipo del campo *c, int* por *AtomicInteger* permite prevenir interferencia entre hilos sin usar la sincronización, como en *AtomicCounter*:

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
```