Unidad 4

DIPLOMATURA EN PROGRAMACION JAVA

Capítulo 8

Threads

Threads

En este módulo

- Creación
- Otra Forma de Crear un Thread
- Tiempos de ejecución de un thread
- Terminando un Thread
- Thread en espera
- Usando la Palabra Clave synchronized
- La Bandera para el Bloqueo de Objetos
- Liberación del indicador de bloqueo
- Interacción entre los threads
- Modelo de control de la sincronización
- Ejemplo de interacción de threads

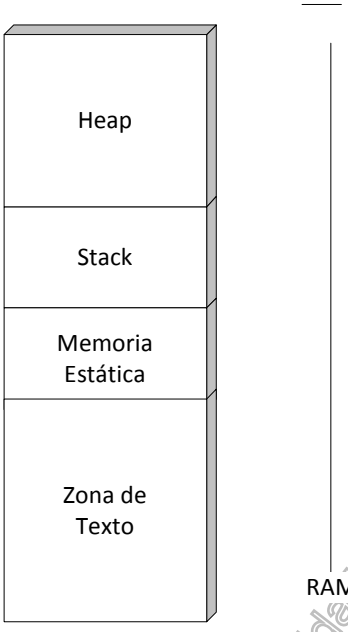
Las Tres Partes de un Thread

Todo programa cuando comienza, se ejecuta sobre un thread principal o "main thread", el cual determina la secuencia de ejecución de las sentencias del programa.

Java posee la habilidad de ejecutar más de un thread dentro de un mismo programa. Esto es, puede crear threads adicionales cuando se lo indique en el código y su creación y manejo es parte del código nativo del lenguaje.

La pregunta fundamental de tener más de una secuencia de procesamiento en un mismo programa es ¿cómo se comporta los elementos del programa en esta situación?

Como se mencionó anteriormente, todo programa respeta la estructura básica que muestra la siguiente figura:



Donde:

- **Zona de Texto**: es el lugar donde se almacenan todas las instrucciones de programa
- **Memoria Estática**: es donde se almacenan las variables globales (según el lenguaje) y las estáticas.
- **Stack**: es el lugar donde se almacenan los parámetros y variables locales de un método. Se agranda o reduce dinámicamente en cada invocación a un método
- **Heap**: es la memoria asignada a un programa que no se encuentra en uso actualmente pero que se puede pedir dinámicamente.

Es claro que los valores que gobiernan la instrucción a ejecutar deberá ser propia de cada thread, ya que cada uno puede estar en ejecutándose en una sentencia diferente de una misma parte del programa.

Por otro lado, el stack también será propio de cada thread por el mismo motivo, ya que cada uno de ellos puede estar en métodos distintos y la secuencia de llamadas puede estar en diferentes etapas.

Sin embargo, la memoria estática y el heap son comunes a todos ellos porque son accesibles desde todo el programa, particularmente, desde cada thread. Esto genera un problema que se analizará posteriormente.

En conclusión, cada thread para ser único necesita manejar diferentes stack e instrucción a ejecutar en el código y tiempo de procesador para hacerlo. Por lo tanto, se resume que cada thread posee como si fueran propios y sólo le pertenecieran:

- Tiempo de CPU
- Datos
- Código

Creación

Conociendo conceptualmente las necesidades de un thread para ejecutarse, el próximo pasa es ver como se crea uno.

Ejemplo

```
package creacion;

public class ThreadHola implements Runnable {
	public void run() {
		int i;
		i = 0;
		while (true) {
			System.out.println("Hola " + i++);
			if (i == 50) {
				break;
			}
		}
	}
}


package creacion;

public class PruebaUnThread {

	public static void main(String args[]) {
		ThreadHola r = new ThreadHola();
		Thread t = new Thread(r);
		t.start();
	}
}
```


Cuando se trabaja en un ambiente multithreading como la JVM, se pueden crear múltiples threads de dos formas:

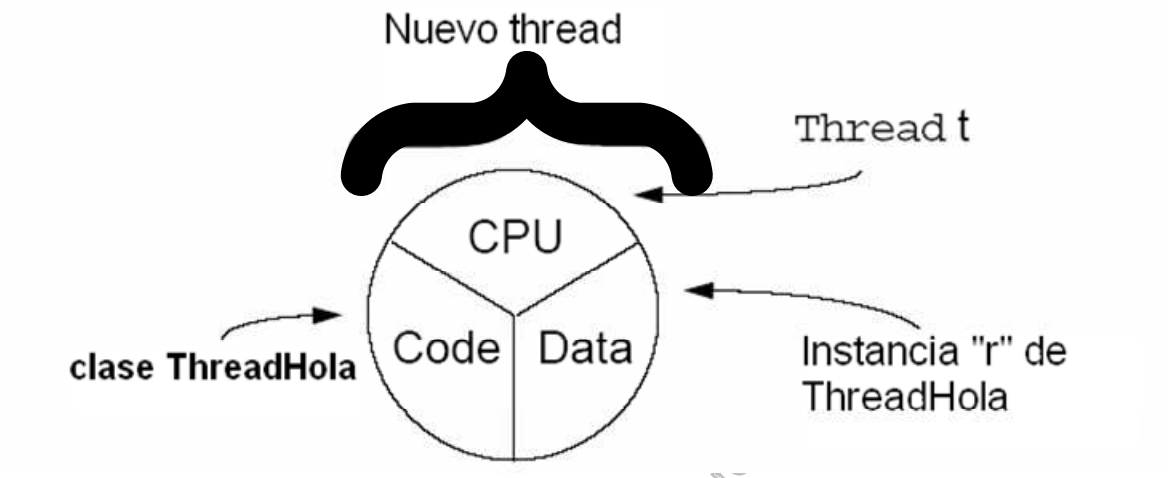
- Multiple threads de la misma instancia de Runnable
- Threads que comparten el mismo código pero en objetos diferentes

Un ejemplo claro de la primera situación es cuando se tiene la instancia r de una clase que implementa la interfaz Runnable:

```
Thread t1 = new Thread(r);
Thread t2 = new Thread(r);
```

La segunda situación es cuando se crean varias instancias de un mismo tipo de objeto y cada cual se invoca en threads diferentes.

Un gráfico que demuestra esquemáticamente como trabaja el nuevo thread creado en el programa de ejemplo es el siguiente:



Otra Forma de Crear un Thread

Los threads pueden implementarse de una forma más sencilla sin la interfaz Runnable heredando directamente de la clase Thread y sobrescribiendo el método run.

Ejemplo

```
package herencia;

public class MiThread extends Thread {
	private boolean ejecutar;

	public void run() {
		while (ejecutar) {
			// Hace algo
			try {
				sleep(100);
			} catch (InterruptedException e) {
				// Se interrumpe el sleep
			}
		}
	}

	public static void main(String args[]) {
		Thread t = new MiThread();
		t.start();
	}
}
```

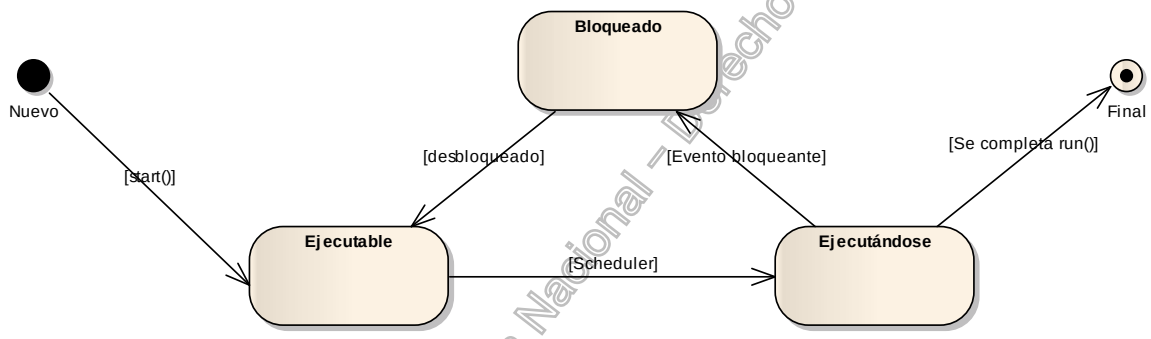
Comienzo de un Thread

Cuando se crea un thread no comienza automáticamente sino que se lo tiene que comenzar explícitamente con el método start().

El llamado a este método coloca a la supuesta CPU definida para el thread en un estado de posible ejecución para ser autorizada a comenzar por el scheduler de la JVM, que a la vez corresponde a su igual en el sistema operativo que se esté ejecutando

Schedule de un Thread

La situación descrita anteriormente se puede analizar mejor en un diagrama de secuencia, el cual refleja el cambio de estado del objeto cuando pasa de un estado "a ejecutar" (Runnable) a un estado en ejecución por designación de tiempo de CPU que realiza el scheduler. El diagrama de estado es el siguiente:



Tiempos de ejecución de un thread

Los threads deben ser pensados en el diseño de un programa como atemporales y asincrónicos. Cada vez que se quiera manejar tiempos con un thread se debe forzar la operación, esto implica invocar métodos especiales para cambiar su funcionamiento atemporal y asíncrono.

Una de estas funciones es sleep, la cual "duerme" al thread por un período de tiempo igual al que se establece en su argumento medido en milisegundos. El thread entra en un estado de espera de ejecución pero **retiene todos los bloqueos y recursos previos a la ejecución del método**. Un ejemplo de esto se puede apreciar en el siguiente código.

Ejemplo

```
package tiempos;

public class Ejecutor implements Runnable {
	public void run() {
		while (true) {
```

```
                        // hace algo
                        // Les da a otros threads una oportunidad
                        try {
                                Thread.sleep(10);
                        } catch (InterruptedException e) {
                                // El sleep de este thread se interrumpió por
                                // otro thread que lo invocó
                        }
                }
        }
}
```

Terminando un Thread

Un thread puede ser forzado a terminar con sólo forzar el fin del método run. Cualquiera sea la forma en que este método finalice, el thread termina su ejecución. Esto puede ser un beneficio o una desventaja si se accede a variables de instancia que intervengan en el procesamiento del método o cualquier otro tipo de suceso que interrumpa la ejecución de run, como una excepción. Por lo tanto, se debe utilizar con precaución.

Ejemplo

```
package tiempos;

public class OtroEjecutor implements Runnable {
        private boolean terminar=false;
        public void run() {
                while ( ! terminar ) {
                        System.out.println("Ciclo en OtroEjecutor");
                } // Limpieza antes que termine run()
        }
        public void finEjecucion() {
                terminar=true;
        }
}


package tiempos;

public class ControladorThread {
        private OtroEjecutor r = new OtroEjecutor();
        private Thread t = new Thread(r);
        public void iniciarThread() {
                t.start();
        }
        public void stopThread() {
                r.finEjecucion(); //Usa la instancia específica de OtroEjecutor
        }
        public static void main(String[] args) {
                int i=0;
                ControladorThread ct = new ControladorThread();
                ct.iniciarThread();
                while (i < 150) System.out.println("Ciclo en main: " + i++);
```

```
            ct.stopThread();
        }
}
```

Control Básico de Threads

Existen varios métodos para el control de threads, algunos de los más importantes son:

- Comprobación de threads:
  - isAlive(): verdadero si el thread se encuentra en ejecución o aún no ha terminado y esta en espera.
  - getName(): retorna el nombre asignado al thread en el constructor. Si no se le asignó uno, retorna el nombre asignado por defecto por la JVM.
- Prioridad de threads:
  - getPriority(): retorna la prioridad del thread.
  - setPriority(): especifica la prioridad del thread. Existen tres constantes en la clase Thread para darle los tres valores típicos al thread de sus 10 posibles:
    - Thread.MAX_PRIORITY (valor 10)
    - Thread.NORM_PRIORITY (valor 5)
    - Thread.MIN_PRIORITY (valor 1)
- Colocando los threads bajo contención:
  - Thread.sleep(): pone al thread en espera tantos milisegundos como indique su argumento.
  - join(): indica que no se termine el thread actual antes de ejecutar al que se invoca el join. Si su argumento tiene un número, indica los milisegundos que espera o la finalización del thread del join, lo que pase primero.
  - Thread.yield(): causa que el thread actual entregue el control que tiene tomado de la CPU.

Thread en espera

Existen mecanismos para bloquear la ejecución de un thread de forma temporal. Luego, es posible reanudarla como si nada hubiese ocurrido. El thread da la apariencia de haber ejecutado una instrucción con mucha lentitud.

Método Thread.sleep()

El método sleep proporciona una forma de detener un thread durante un periodo de tiempo. Tener presente que el thread no necesariamente reanuda su ejecución en el momento en que finaliza el periodo de espera. La razón es que puede haber otro thread en ejecución en ese momento y no se detendrá a menos que se produzca una de las siguientes situaciones:

- El hilo que se reactiva tiene mayor prioridad.
- El hilo en ejecución se bloquea por alguna otra razón.

Método join

El método join hace que el thread actual espere hasta que finalice el thread en el que se ha realizado la llamada a join.

Por ejemplo, se puede suponer que hay que generar múltiples threads para hacer un determinado trabajo y continuar con el siguiente paso sólo después de que todos ellos completos. Hay que especificarle al thread principal que espere. El punto clave es utilizar el método Thread.Join ().

Ejemplo

```
package esperas;
import java.util.ArrayList;

public class VerificarThreads {

	private ArrayList<String> nombresDeThreads = new ArrayList<String>();

	public static void main(String[] args) {
		int cantidadThreads = 10;
		VerificarThreads test = new VerificarThreads();
		test.testThread(cantidadThreads);
		System.out.println(test.nombresDeThreads);
		System.out.println("Esto se imprime al final");
		System.out.println(Thread.currentThread().getName());
	}

	private void testThread(int numOfThreads) {
		Thread[] threads = new Thread[numOfThreads];
		for (int i = 0; i < threads.length; i++) {
			threads[i] = new esperas.VerificarThreads.MiThread();
			threads[i].start();
		}
		System.out.println("Esto se imprime antes");
		System.out.println(Thread.currentThread().getName());

		for (int i = 0; i < threads.length; i++) {
			try {
				threads[i].join();
			} catch (InterruptedException ignorar) {
			}
		}
	}

	public class MiThread extends Thread {

		@Override
		public void run() {
			for (int i = 0; i < 1000000; i++) {
				i = i + 0;
			}
			nombresDeThreads.add(getName());
			System.out.println(getName());
```

```
            }
      }
}
```

La salida del programa cuando se ejecuta es

```
Esto se imprime antes
main
Thread-1
Thread-4
Thread-3
Thread-2
Thread-7
Thread-6
Thread-8
Thread-9
Thread-5
Thread-0
[Thread-1, Thread-3, Thread-4, Thread-0, Thread-2, Thread-5, Thread-7, Thread-6,
Thread-8, Thread-9]
Esto se imprime al final
main
```

El orden en que los threads se ejecutan es aleatorio, lo cual es como se espera. También tener en cuenta que se utilizan dos ciclos, el primero en crear e iniciar cada thread, y el segundo para "unir" cada thread. Si cada uno se unió después del comienzo, el efecto es que estos hilos se ejecuten secuencialmente, sin la concurrencia deseada porque cada uno espera al anterior para terminar.

Ejemplo

```
package esperas;
import java.util.ArrayList;

public class VerificarThreadsSecuenciales {

      private ArrayList<String> nombresDeThreads = new ArrayList<String>();

      public static void main(String[] args) {
            int cantidadThreads = 10;
            VerificarThreadsSecuenciales test = new
                  VerificarThreadsSecuenciales();
            test.testThread(cantidadThreads);
            System.out.println(test.nombresDeThreads);
            System.out.println("Esto se imprime al final");
            System.out.println(Thread.currentThread().getName());
      }

      private void testThread(int numOfThreads) {
            Thread[] threads = new Thread[numOfThreads];
            for (int i = 0; i < threads.length; i++) {
                  threads[i] = new
                        esperas.VerificarThreadsSecuenciales.MiThread();
                  threads[i].start();
                  try {
```

```
                        threads[i].join();
                    } catch (InterruptedException ignorar) {
                    }
            }
            System.out.println("Esto se imprime antes");
            System.out.println(Thread.currentThread().getName());
      }

      public class MiThread extends Thread {
            public void run() {
                  for (int i = 0; i < 1000000; i++) {
                        i = i + 0;
                  }
                  nombresDeThreads.add(getName());
                  System.out.println(getName());
            }
      }
}
```

La salida del programa es

```
Thread-0
Thread-1
Thread-2
Thread-3
Thread-4
Thread-5
Thread-6
Thread-7
Thread-8
Thread-9
Esto se imprime antes
main
[Thread-0, Thread-1, Thread-2, Thread-3, Thread-4, Thread-5, Thread-6, Thread-7,
Thread-8, Thread-9]
Esto se imprime al final
main
```

Si no se usa ninguna join en absoluto, nombresDeThreads, cuando se imprime, puede estar vacío o parcialmente lleno, ya que el thread principal continuará cuando tenga la oportunidad. El thread principal aún puede esperar, en el último paso, a que todos los threads se completen, antes de salir de la JVM. El código del método quedaría

```
      private void testThread(int numOfThreads) {
            Thread[] threads = new Thread[numOfThreads];
            for (int i = 0; i < threads.length; i++) {
                  threads[i] = new esperas.VerificarThreadsSinJoin.MiThread();
                  threads[i].start();
            }
            System.out.println("Esto se imprime antes");
            System.out.println(Thread.currentThread().getName());
      }
```

La salida para el funcionamiento de 10 threads puede ser:

```
Esto se imprime antes
main
[]
Esto se imprime al final
main
Thread-3
Thread-6
Thread-7
Thread-1
Thread-0
Thread-4
Thread-9
Thread-8
Thread-2
Thread-5
```

Notar la salida en el tercer renglón de [] para el ArrayList vacío.

También se puede llamar al método join utilizando un valor de tiempo de espera expresado en milisegundos. Por ejemplo:

```
void join(long timeout);
```

En este ejemplo, el método join interrumpe el hilo actual durante los milisegundos indicados en timeout o bien hasta que finaliza el hilo en el que se ha efectuado la llamada.

Método Thread.yield()

El método Thread.yield() se utiliza para dar a otros threads ejecutables la oportunidad de ejecutarse. Si hay otros threads ejecutables, yield sitúa el thread que hace la llamada en el grupo de ejecutables y permite a otro thread del grupo ejecutarse. Si no hay otros threads ejecutables, yield no hace nada.

La llamada sleep da a otros threads de menor prioridad la opción de ejecutarse. El método yield ofrece a otros hilos ejecutables la oportunidad de ejecutarse.

Ejemplo

```
package esperas;

public class ThreadsConYield extends Thread {

        private int total = 5;

        private static int contadorThread = 0;

        public ThreadsConYield() {
          super("" + ++contadorThread);
          start();
        }
```

```
    public String toString() {
      return "#" + getName() + ": " + total;
    }

    public void run() {
      while (true) {
        System.out.println(this);
        if (--total == 0){
              try {
                          sleep(10);
                    } catch (InterruptedException e) {
                          e.printStackTrace();
                    }
          return;
        }
        yield();
      }
    }

    public static void main(String[] args) {
      for (int i = 0; i < 5; i++)
        new ThreadsConYield();
    }
}
```

Elección de una Forma

Ante las dos posibilidades de creación de threads, mediante la interfaz Runnable o heredando, se puede comparar como guía cada caso para implementar el más apropiado en cada situación de diseño:

- **Implementando Runnable**:
  - *Diseño mejor orientado a objetos*: la clase Thread es estrictamente una CPU virtual encapsulada y, como tal, sólo debería generar subclases (ampliarse) cuando se modifique o amplíe el comportamiento de ese modelo de CPU. Por este motivo, y por la importancia que tiene diferenciar entre la CPU, el código y las partes de datos de un hilo en ejecución.
  - *Mantiene la herencia simple*: dado que la tecnología Java sólo admite el modelo de herencia sencilla, no es posible ampliar otras clases, como Applet, si ya se ha ampliado Thread. En algunas situaciones, esto obliga a adoptar la práctica de implementar Runnable.
  - *Es más consistente*: Puesto que hay momentos en los que se está obligado a implementar Runnable, es preferible mantener la coherencia y hacerlo siempre de esa forma.
- **Heredando de Thread**:
  - Código más simple: sólo hay que rescribir run y como se está en la misma clase suele ser más sencillo el código.

Usando la Palabra Clave synchronized

Todavía queda por tratar el tema de la memoria compartida. Existen situaciones en las cuales dos o más threads compiten por acceder a un espacio de memoria en común. Cuando esta situación puede provocar inconvenientes en el procesamiento normal de cada uno de ellos, se debe bloquear la memoria para que ninguno cause inconvenientes al otro.

Para exponer el problema, se plantea la siguiente situación en la cual existe una pila manejada como un vector y se utiliza una variable de instancia para controlar el elemento a acceder dentro de él. Si esta variable es incrementada o decrementada en los métodos de extracción o escritura más allá del valor esperado, puede ocasionar que la pila quede inconsistente.

Ejemplo

```
package sincronizar;

public class PilaSinSincronizar {
	int indice = 0;
	char [] datos = new char[6];
	public void poner(char c) {
		datos[indice] = c;
		indice++;
	}
	public char sacar() {
		indice--;
		return datos[indice];
	}
}
```

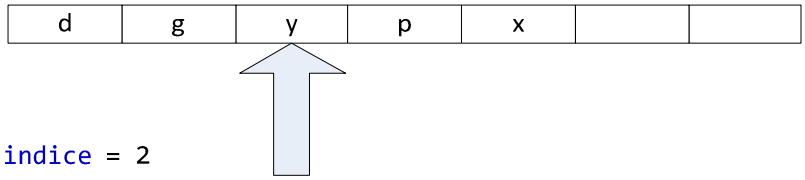
Puede suceder que para un mismo objeto del tipo PilaSinSincronizar se invoque en threads independientes los métodos poner y sacar. Esto puede ocasionar que se incremente (o lo contrario) el valor del índice más allá de que debería hacerse. Se necesita colocar una “bandera” para que un thread indique a cualquier otro que está trabajando sobre la variable que mientras el la acceda ningún otro podrá hacerlo.

El comportamiento de este código exige que el valor de índice contenga el subíndice de vector de la siguiente celda vacía de la pila. El uso del mecanismo pre decremento, pos incremento genera esta información.

Suponiendo que ahora que dos threads contienen una referencia a **una sola instancia** de esta clase. Uno de los threads introduce datos en la pila y el otro, de forma más o menos independiente, retira datos de la pila. En principio, los datos se agregan y suprimen correctamente. No obstante, existe un problema potencial.

Si el thread “Thread 0” está agregando caracteres y el thread “Thread 1” los está suprimiendo y el thread “Thread 0” acaba de escribir un carácter, pero aún no ha incrementado el contador del índice. Entonces, por alguna razón, dicho thread es sustituido por otro que se apropia del derecho

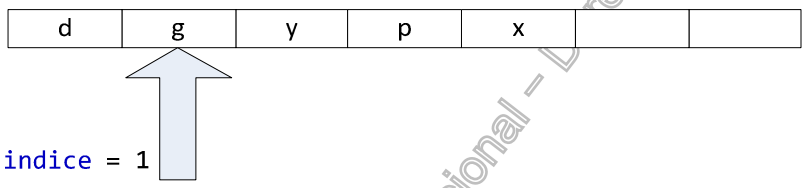
de ejecución. En este momento, el modelo de datos representado por el objeto deja de ser coherente.



Para mantener la coherencia, es necesario que sea indice = 3 o que el carácter no se haya agregado aún.

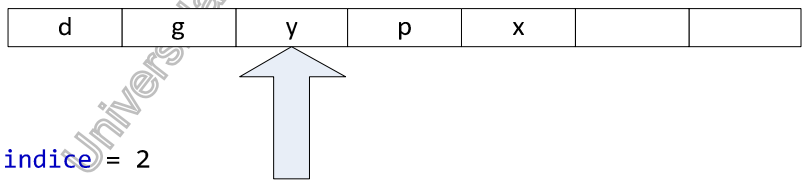
Si el "Thread 0" reanuda la ejecución, puede que no se produzcan errores, pero suponiendo que el "Thread 1" estaba esperando a suprimir un carácter mientras que el "Thread 0" está esperando a tener otra oportunidad de ejecutarse, "Thread 1" tiene la ocasión de suprimir un carácter.

Se produce una situación de falta de coherencia de los datos de entrada del método poner y, sin embargo, dicho método procede a reducir el valor del índice.



En la práctica, esto hace que no se tenga en cuenta el carácter **y**. A continuación, devuelve el carácter g. Hasta ahora, el comportamiento ha sido como si la letra **y** no se hubiese agregado, por lo que es difícil detectar que hay un problema. Pero observar lo que ocurre cuando el thread original, "Thread 0", reanuda la ejecución.

El "Thread 0" reinicia la ejecución en el punto donde la dejó, en el método sacar, y procede a incrementar el valor del índice. Ahora tenemos lo siguiente:



Esta configuración implica que el carácter **g** es válido y que la celda que contiene la **y** es la siguiente celda vacía. En otras palabras, **g** se lee como si se hubiese colocado dos veces en la pila y la letra **y** no aparece nunca.

Esto es un ejemplo de un problema general que surge cuando hay varios threads que comparten los datos a los que acceden. Es preciso disponer de un mecanismo que garantice que esos datos compartidos mantengan la integridad antes de que cualquiera de los threads empiece a utilizarlos para una determinada tarea.

Una forma de hacerlo sería impedir que el thread “Thread 0” interrumpa su ejecución hasta que finalice la sección crítica de código. Este planteamiento es habitual en programación en código máquina de bajo nivel, pero suele ser inadecuado en sistemas multiusuario.

Otro enfoque, que es el utilizado por la tecnología Java, es proporcionar un mecanismo para tratar los datos con delicadeza. Este enfoque permite que un thread se trate atomizado y sea capaz de acceder a los datos con independencia de que lo interrumpan mientras está realizando el acceso.

La Bandera para el Bloqueo de Objetos

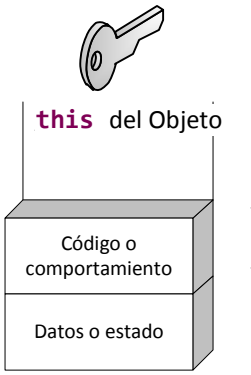
Java maneja las situaciones de memoria compartida a través de banderas de bloqueo internas, las cuales sirven para que cualquier thread que quiera acceder a dicha variable de instancia deba consultar si la misma es accesible o esta **bloqueada**.

Estas situaciones se dan sólo en programación multithreading y para manejarla el lenguaje incorpora la palabra reservada synchronized, la cual permite la interacción con la bandera de bloqueo.

Ejemplo

```
package sincronizar;

public class PilaSincronizada {
	int indice = 0;
	char[] datos = new char[6];

	public void poner(char c) {
		synchronized (this) {
			datos[indice] = c;
			indice++;
		}
	}

	public char sacar() {
		synchronized (this) {
			indice--;
			return datos[indice];
		}
	}
}
```

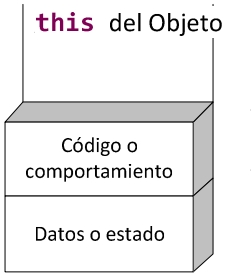
El siguiente gráfico muestra dicha situación:



Thread antes de `synchronized (this)`

```
public void poner(char c) {
        synchronized (this) {
                datos[indice] = c;
                indice++;
        }
}
```

El bloque de código asociado a **synchronized** indica cuando el thread deberá consultar si la variable que pertenece a la memoria compartida puede ser accedida y, en caso de ser así, tendrá que bloquearla antes de ejecutar el código en el bloque. Esto le permite interactuar libremente con la variable con la seguridad que ningún otro thread modificará el valor en ella en tanto no se termine la ejecución del código definido en su bloque de sentencias. Para ello, cuando ingresa al bloque definido por **synchronized**(**this**), intenta adquirir un bloqueo sobre las variables de instancia utilizadas dentro del bloque de sentencias asociado.

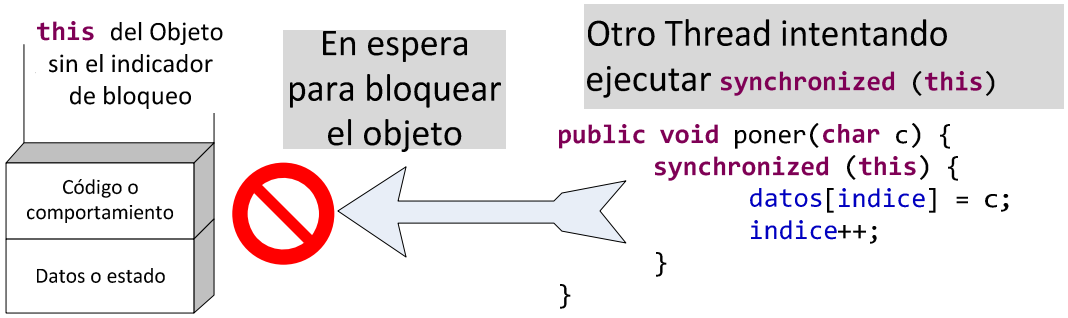


Thread después de `synchronized (this)`

```
public void poner(char c) {
        synchronized (this) {
                datos[indice] = c;
                indice++;
        }
}
```


Es importante tener en cuenta que esta acción no protege los datos. Si el método sacar del objeto que posee los datos compartidos no está protegido por **synchronized** y otro thread hace una llamada a sacar, sigue existiendo el riesgo de perder la coherencia de los datos. Todos los métodos que acceden a los datos deben sincronizarse con respecto al mismo indicador de bloqueo para que éste sea efectivo.

En la siguiente figura se observa lo que ocurre si el método sacar está protegido por la sentencia **synchronized** y otro thread intenta ejecutar el método sacar de un objeto mientras el thread original retiene el indicador de bloqueo del objeto sincronizado.



Cuando el thread quiere ejecutar la primera sentencia dentro del bloque asociado a **synchronized**(**this**), trata de conseguir el indicador de bloqueo que tiene el objeto actualmente en ejecución cuya referencia se encuentra en **this**. Como no lo consigue, no puede continuar la ejecución. Seguidamente, el thread se une a un grupo de éstos, que se encuentran en las mismas condiciones en estado de espera, que están asociados al indicador de bloqueo de dicho objeto. Cuando el indicador retorna y queda disponible en el objeto, se entrega el bloqueo a uno de los threads que estén esperándolo y es ese thread quien continúa con la ejecución.

Liberación del indicador de bloqueo

Si un thread está esperando el indicador de bloqueo de un objeto, no puede reanudar su ejecución hasta que el indicador esté disponible. Por tanto, es importante que el hilo que retiene el bloqueo devuelva el indicador cuando deja de necesitarlo.

En ese momento, el objeto que tiene la memoria compartida recobra el indicador de forma automática. Cuando el thread que retiene el bloqueo pasa el final del bloque de código de la sentencia **synchronized** con la que se obtuvo el bloqueo, éste se libera. Java garantiza que el bloqueo siempre se devuelva de forma automática, incluso aunque una excepción, una sentencia de interrupción o una sentencia de retorno transfieran la ejecución del código a otra parte situada fuera de un bloque sincronizado. Asimismo, si un thread ejecuta bloques anidados de código que están sincronizados con respecto al mismo objeto, el indicador de ese objeto se libera correctamente al salir del bloque externo de la secuencia y se hace caso omiso del primer bloque interno.

Estas reglas simplifican el manejo de bloques sincronizados bastante más que las utilidades equivalentes en algunos otros sistemas

Uso de **synchronized**: conclusiones

El mecanismo de sincronización sólo funciona si todo el acceso a los datos “sensibles” se produce dentro de los bloques sincronizados.

Los datos protegidos por los bloques sincronizados deberían marcarse como **private**. A esto se lo denomina memoria compartida. Si no se hace así, será posible acceder a los datos sensibles desde

código situado fuera de la definición de la clase. Semejante situación permitiría a otro código sortear la protección y dañar los datos durante el tiempo de ejecución.

Un método compuesto enteramente de código perteneciente a un bloque sincronizado con respecto a la instancia **this** podría incluir la palabra clave **synchronized** en la declaración del bloque. Los dos fragmentos de código siguientes son equivalentes:

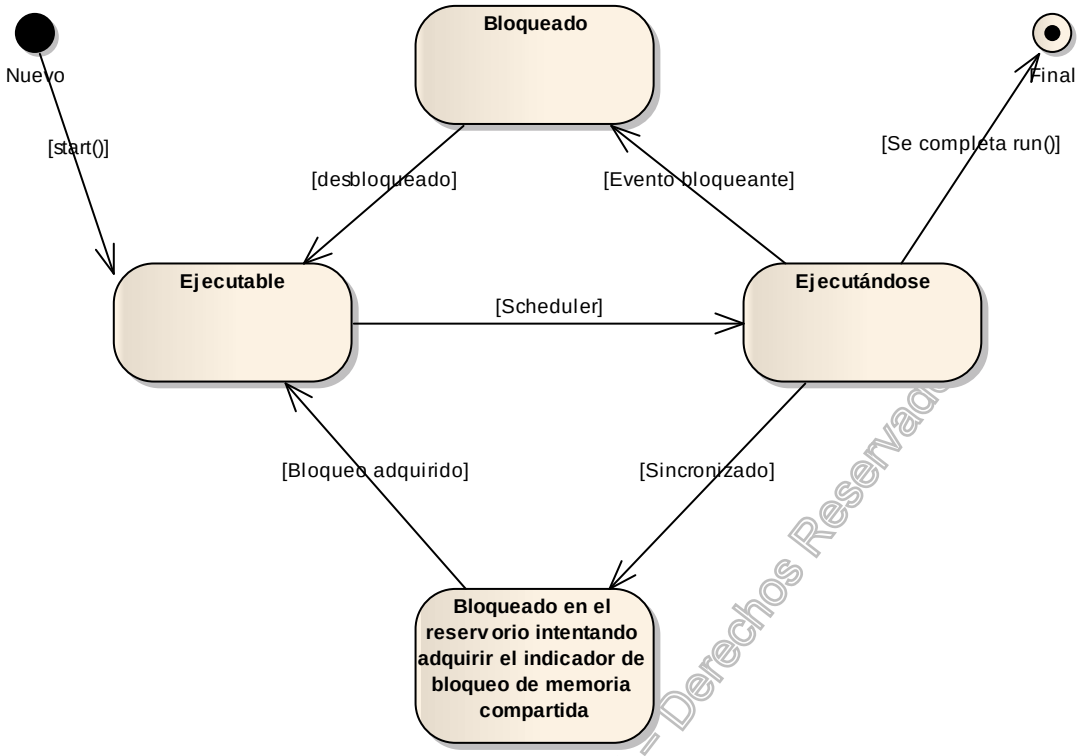
```
public void poner(char c) {
	synchronized (this) {
		datos[indice] = c;
		indice++;
	}
}


public synchronized void poner(char c) {
		datos[indice] = c;
		indice++;
}
```

¿Por qué usar una técnica en lugar de la otra? Si se utiliza **synchronized** como modificador de un método, todo el método se convierte en un bloque sincronizado. Esto podría hacer que el indicador de bloqueo quedase retenido más tiempo del necesario.

No obstante, marcar el método de esta manera permite a los usuarios del método saber, a partir de la documentación generada por la utilidad javadoc, que se está produciendo la sincronización. Esto puede ser importante al diseñar código para evitar el interbloqueo (que se describe en la siguiente sección). La utilidad javadoc propaga el modificador **synchronized** por los archivos de documentación que genera, pero no puede hacer los mismo con la sentencia **synchronized**(**this**), que se encuentra en el interior del bloque del método.

Diagrama de Estado de Sincronización de un Thread

Cuando se ejecuta un bloque de sincronizado, todo thread que queda a la espera de la oportunidad de ejecutarse mantiene una referencia en un reservorio (en inglés se lo suele llamar “pool”) que es accesible por el scheduler. Cuando se libera el indicador de bloqueo del objeto que posee la memoria compartida, los threads que estaban aguardando pasan a un estado de “ejecutable”. El scheduler selecciona uno de ellos y le permite el acceso al indicador de bloqueo y le concede tiempo de CPU para ejecutarse. Un diagrama de estado que refleja la situación es el siguiente:



Interbloqueo (deadlock)

En programas donde hay varios threads compitiendo por acceder a diferentes recursos, puede producirse una situación conocida como interbloqueo. Se produce cuando un thread está esperando a adquirir un bloqueo retenido por otro, pero el otro threadestá esperando a su vez otro bloqueo retenido por el primero. En esta situación, ninguno de los dos puede proseguir la ejecución hasta que el otro haya pasado el fin de su bloque `synchronized`. Como ninguno puede proseguir, ninguno puede pasar el final de su bloque y su ejecución no termina nunca.

Java ni detecta ni intenta evitar esta situación. Es responsabilidad del programador asegurarse de que no pueda producirse un interbloqueo.

Se pueden seguir dos simples reglas para evitar los interbloqueos que suelen ser difíciles de seguir cunado se diseña o se mantienen los programas y son las siguientes:

- *Si se dispone de múltiples objetos a los que se quiere proporcionar acceso sincronizado, tomar una decisión global sobre el orden en el que obtendrá los bloqueos y respetar dicho orden a lo largo del programa.*
- *Liberar los bloqueos en orden inverso a aquel con el que se han obtenido.*

Interacción de los hilos: wait y notify

La clave para manejar un buen diseño de threads es no depender de su tiempo de ejecución, o sea, no pensarlos como trabajos secuenciales sino como procesamiento paralelo y no

sincronizado. Cuando se quiere procesar threads en forma secuencial se debe recurrir a llamados de métodos como join, pero en ese punto cabe el planteo si no se puede solucionar el problema con llamados a métodos simples. Sin embargo, se presentan situaciones en las cuales dos o más threads pueden necesitar cierto nivel de sincronía y este a su vez, sea independiente del thread principal de ejecución que los crea. Este es el caso en el cual los trabajos que realizan se relacionan de alguna manera y podría ser necesario programar algunas interacciones entre ellos.

En el desarrollo se presenta a menudo una situación en la cual una aplicación consulta a otra si un determinado suceso se ha producido. Este es el ejemplo en el que dos programas completamente disjuntos interaccionan. Por ejemplo, el driver del sistema de archivos de un sistema operativo se encarga de manejar todos los eventos que permiten gestionar documentos, directorios, etc., y una aplicación quiere "enterarse" si en un determinado directorio se ha cambiado la cantidad o el contenido de los archivos. Esta tendría supuestamente dos caminos a seguir: manejar por si mismas los archivos de dicho directorio o consultar a intervalos de tiempo si se produjeron cambios. La solución obvia siempre es la segunda porque la primera es una función del sistema operativo.

¿Cuál es el problema de este ejemplo? Qué los procesos son disjuntos, no se encuentran regidos por el mismo programa, por lo cual la única solución en consultar al otro programa por espacios de tiempo.

Se puede dar otra situación diferente y es que el programa maneje en su interior ambos procesos al mismo tiempo pero dichos procesos sean disjuntos. Por ejemplo, se quiere manejar la lectura y escritura a un archivo de bloques de información que puede ser actualizada aleatoriamente dentro del mismo. Es claro que si un bloque se actualiza mientras otro lee se produciría inconsistencias entre los bloques leídos y la información que realmente contiene el archivo porque mientras se está leyendo se puede estar modificando. Sin embargo del análisis también puede surgir que no se debe bloquear a todo el programa mientras se realiza una operación de entrada – salida porque éste necesita seguir procesando.

Dentro de las posibles soluciones hay una en particular que parece la óptima: seguir procesando y crear uno o más threads para leer, uno o más para escribir, que ambos tipos sean independientes del thread que los crea y que los de un tipo no puedan realizarse hasta que los del otro termine. La última afirmación indica indirectamente una situación de sincronía en la cual tipo de thread debe **avisar** a los del otro tipo que pueden iniciar su procesamiento.

Como se puede apreciar esto indica sincronía entre threads, pero no secuencialidad, ya que **cualquier thread de un tipo se puede ejecutar cuando cualquier thread del otro tipo indica que estás disponible los elementos que se desean procesar** (en este caso, el archivo a leer).

Si ambos tipos de threads compiten por obtener los recursos para esta operación, como ser el lugar de memoria donde se almacena lo leído, la información del tamaño del archivo, etc., donde puedan ocurrir operaciones de lectura o escritura, si uno de los de un tipo obtiene una parte de los recursos y otro de los otro tipo otra parte de **los mismos recursos necesarios, ambos se bloquean**

entre si esperando a que el otro libere recursos para bloquearlos y así lograr el acceso. Esta es una situación clara de interbloqueo y se debe evitar.

Por lo tanto, se debe resolver ambos problemas a la vez, que un tipo de thread no corrompa la información que esta usando los del otro tipo (situación que se resuelve sincronizando la memoria compartida) y que un tipo de thread se "entere" cuando el otro tipo deja la información y recursos a manejar en un estado consistente para su uso. En pocas palabras, los threads de un tipo esperan la notificación de los del otro tipo para procesar en forma confiable su tarea.

Interacción entre los threads

Por lo que se explicó en la sección anterior, es necesario tener un mecanismo que permita a los threads interaccionar entre si para que cuando se realiza algún tipo de procesamiento se habilite la capacidad de realizar otro procesamiento en un thread diferente. Esto va más allá de la utilización de recursos compartidos como la memoria. Esto significa que los recursos deben poseer cierto "estado" para otro tipo de thread pueda realizar sus tareas.

> *A este tipo de procesamiento en el cual dos o más tipos diferentes de procesos pueden trabajar sobre los mismos recursos para cambiar su estado se los denomina procesos concurrentes.*

Métodos wait y notify

Cuando existe concurrencia, los procesos se encuentran en una estado denominado Rendezvous, la cual es una primitiva de sincronización asimétrica que permite a dos procesos concurrentes llamados, el solicitante y el invocado, intercambiar datos de forma coordinada. El proceso que solicita el rendezvous debe esperar en un punto de rencuentro hasta que el proceso invocado llegue allí. Igualmente el proceso invocado puede llegar al rendezvous antes que el solicitante y debe esperar que este llegue al punto de encuentro para poder continuar procesando. La imagen de esperar en un punto de encuentro corresponde a colocar un proceso en espera inactiva hasta que la cita se cumpla. Durante el rendezvous los procesos pueden intercambiar datos.

Los datos intercambiados corresponden a parámetros de una llamada (desde el solicitante hacia el proceso invocado) y a resultados de una llamada (desde el proceso invocado hacia el solicitante), sin necesidad de almacenamiento intermedio.

La clase java.lang.Object proporciona dos métodos para posibilitar la comunicación entre threads: wait y notify. Si uno hace una llamada a wait referida a un objeto de comunicación (Rendezvous) x, ese thread detiene su ejecución hasta que otro genera una llamada a notify para ese mismo objeto x.

En la explicación anterior, un proceso que lee el archivo debe esperar (wait) a que un proceso que este modificándolo termine y avise (notify).

Pero tanto para esperar, como para notificar, se debe realizar cuando la tarea este completa y no a medio hacer, con lo cual estas declaraciones deben estar dentro de un bloque sincronizado. Para el ejemplo anterior, el bloque sincronizado estaría dado, por ejemplo, por dos métodos del tipo `synchronized` leer() y `synchronized` modificar(). En el primero se colocaría un wait para esperar que el archivo este disponible para leer y en el segundo un notify para indicar que el proceso de actualización finalizó. De esta manera se pueden manejar tantas lecturas y modificaciones como se quieran en procesos independientes concurrentes al recurso (el archivo).

Descripción de los reservorios

Cuando un hilo ejecuta código sincronizado que contiene una llamada wait referida a un determinado objeto, ese hilo se coloca en el grupo de espera para ese objeto (reservorio o "pool" en inglés). Se debe tener en cuenta que, por lo general, el thread que esta ejecutando la tarea está declarado en un objeto diferente a aquel que posee el bloque sincronizado donde se invoque a wait o notify. Asimismo, el thread que llama a wait libera el indicador de bloqueo que posee de ese objeto de forma automática. Es posible llamar a diferentes métodos wait.

- wait()
- wait(`long` timeout)

Cuando se ejecuta una llamada a notify referida a un determinado objeto (el cual es que posee los recursos compartidos que son accedidos desde un bloque sincronizado), un thread arbitrario se traslada desde el grupo de espera a un grupo de bloqueo para dicho objeto, donde los threads permanecen hasta que se libera el bloqueo de los recursos tomado por el bloque sincronizado que posee dicho objeto. El método notifyAll traslada todos los threads del grupo de espera de ese objeto al grupo de bloqueo. Un thread sólo puede obtener el indicador de bloqueo del objeto desde el grupo de bloqueo, lo que le permite continuar ejecutándose en el punto donde se encontraba al llamar a wait.

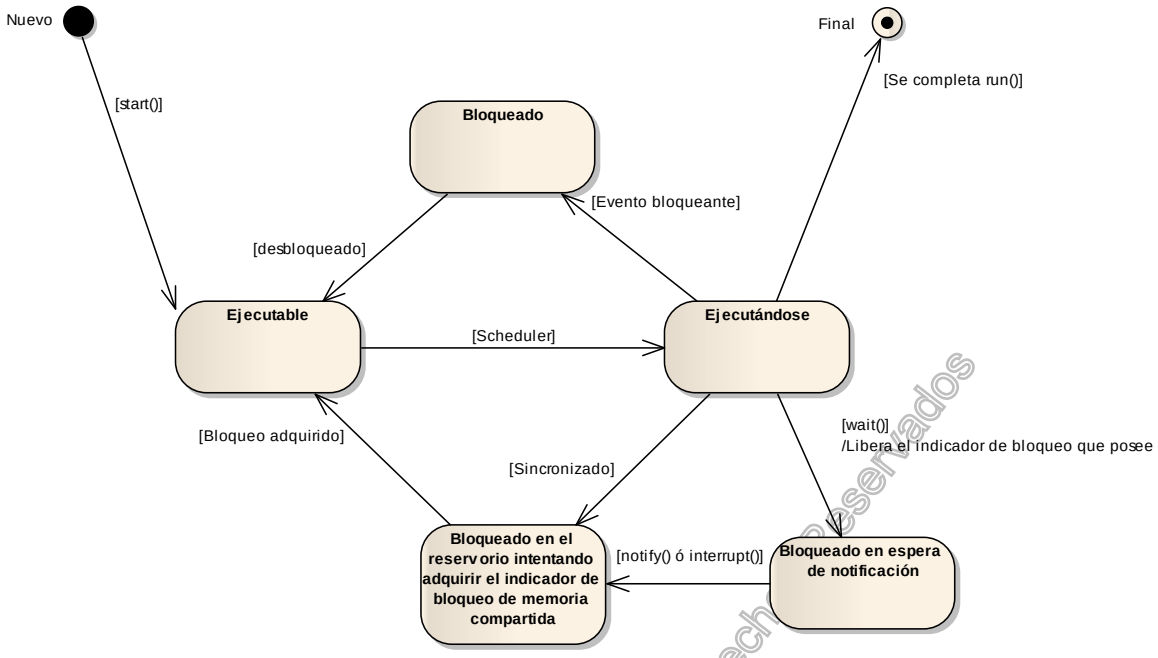
En muchos sistemas que implementan el mecanismo wait-notify, el hilo que se reactiva es el que haya esperado durante más tiempo. La tecnología Java no garantiza que esto sea así.

Es posible hacer una llamada a notify aunque no haya ningún thread esperando. Si se llama al método notify con referencia a un objeto cuando no hay ningún thread en el grupo de espera aguardando el indicador de bloqueo de ese objeto, la llamada no tiene ningún efecto.

Las llamadas a notify no se almacenan para una ejecución posterior nunca. Se ejecutan sólo en el momento de la invocación y tienen efecto sólo si se dan las condiciones antes explicadas respecto de un wait.

Estados de los hilos

El grupo de espera también es un estado especial de los threads. En la siguiente figura se puede ver el diagrama de transición de los estados de un thread hasta que es seleccionable para el recolector de basura.



Modelo de control de la sincronización

La coordinación entre dos threads que necesitan acceso a datos comunes puede ser compleja. Es necesario garantizar que ningún thread deje datos compartidos en un estado incongruente cuando existe la posibilidad de que otro acceda a dichos datos. También es importante asegurar que el programa no producirá situaciones de interbloqueo porque uno no pueda liberar un bloqueo que otro está esperando.

En el ejemplo del taxi, el código se basaba en la existencia de un objeto de comunicación, el taxi, sobre el que se ejecutaban los métodos wait y notify. Si alguien estuviese esperando un autobús, se necesitaría un objeto autobús independiente al que aplicar la llamada a notify.

Recuerde que todos los hilos del mismo grupo de espera deben recibir la oportuna notificación del objeto que controla ese grupo. No diseñe nunca código donde los hilos deban recibir notificación sobre situaciones diferentes en el mismo grupo de espera.

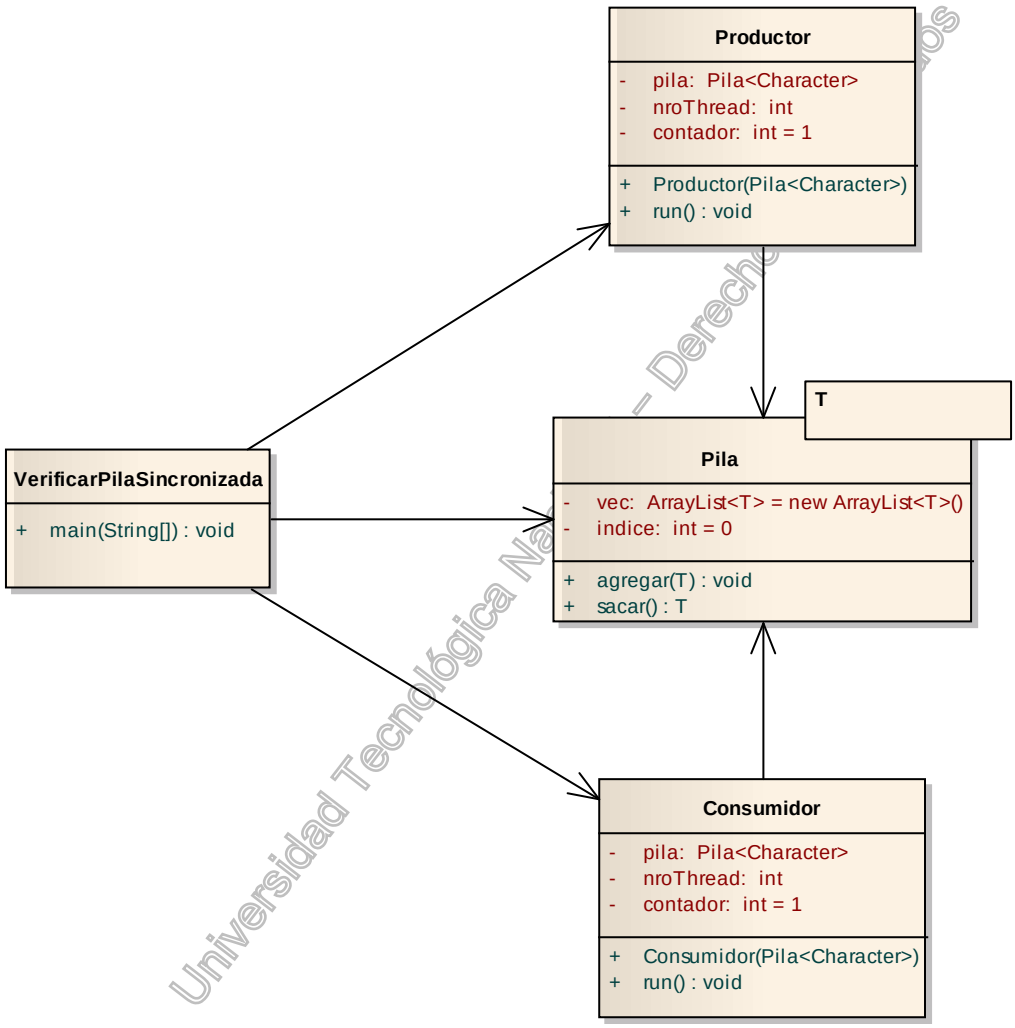
Ejemplo de interacción de threads

El código de esta sección contiene un ejemplo de interacción de hilos que muestra el uso de los métodos wait y notify para resolver un problema clásico de productor-consumidor. Para eso se va a utilizar un ejemplo mostrado en el módulo de colecciones, la pila genérica.

La idea parte de extender dicho ejemplo para que maneje el acceso de un objeto de tipo pila a la memoria compartida cuando invocan sus servicios threads concurrentes a ésta.

Los servicios de un objeto del tipo pila están brindados por los métodos poner y sacar. Ambos acceden al ArrayList definido en la clase para operar con él sacando y poniendo elementos. Si un thread accede al método sacar mientras que otro accede al método poner concurrentemente sobre esta pila, se está en presencia del caso mencionado anteriormente y debe manejarse para que ambos threads no invadan la memoria compartida simultáneamente.

Básicamente, el ejemplo planteado a partir de la clase Pila diseñada para los módulos anteriores tiene un diseño como el que se muestra a continuación:



Para resolver el problema de concurrencia se inicia declarando los métodos que acceden a la memoria compartida como bloques sincronizados. Esto implica que cada vez que se llame a uno de ellos, deberán competir para obtener un indicador de bloqueo cuando se encuentren llamando a los servicios de un mismo objeto.

Nota: *Cuando una clase declara bloques sincronizados en sus servicios determina que diferentes threads que acceden los servicios de un mismo objeto de su tipo deban obtener un indicador de bloqueo para acceder a las variables de instancia de éste. A esto se lo suele denominar "a salvo de los threads (thread safe)" porque ninguno puede invadir la memoria compartida cuando otro la está accediendo*.

La clase Pila

El método sacar

Para modificar la clase Pila de manera que pueda prestar sus servicios cuando son requeridos concurrentemente, se debe analizar y rediseñar el código existente para que proteja las variables de instancia del objeto ya estas serán la "memoria compartida". El código original del método sacar se presenta a continuación.

```
public T sacar() throws ExcepcionPila {
	if (indice == 0)
		throw new ExcepcionPila("Pila vacía");
	T t = vec.get(indice);
	vec.remove(indice);
	indice--;
	return t;
}
```

La primera modificación se nota rápidamente, el método debe estar sincronizado para proteger la memoria compartida.

```
public synchronized T sacar() throws ExcepcionPila {
	if (indice == 0)
		throw new ExcepcionPila("Pila vacía");
	T t = vec.get(indice);
	vec.remove(indice);
	indice--;
	return t;
}
```

Sin embargo, esto no alcanza para evitar que se intente consumir, por ejemplo, un carácter cuando la pila está vacía. Por lo mencionado anteriormente de debe colocar al objeto que requiere el servicio en espera hasta que "algo" informe que se encuentran elementos para consumir disponibles. Como se verá posteriormente, el encargado de este aviso es el thread que produce elementos que se colocan en la pila.

El método sacar de la pila debe estar sincronizado por dos motivos. En primer lugar, extraer un elemento de la pila afecta al ArrayList de datos compartidos. En segundo lugar, la llamada a wait debe estar incluida dentro de un bloque que esté sincronizado con respecto al objeto actualmente en ejecución, que está representado por la palabra **this**. El nuevo código modificado asume el siguiente formato.

```
public synchronized T sacar() throws ExcepcionPila {
```

```
            try {
                  this.wait();
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
            if (indice == 0)
                  throw new ExcepcionPila("Pila vacía");
            T t = vec.get(indice);
            vec.remove(indice);
            indice--;
            return t;
      }
```

La llamada a wait se coloca en un bloque try-catch porque una llamada a interrupt puede poner fin al tiempo de espera del thread, esto es, cualquier situación que interrumpa la ejecución del thread que actualmente llamo a wait.

Sin embargo, el código aún tiene un problema a resolver. Si, por algún motivo (como una interrupción) el thread se reactiva y descubre que la pila sigue estando vacía, debería volver a entrar en estado de espera. Una forma de solucionar esto es verificar el tamaño de la pila para que, en caso de estar vacía, vuelva a invocarse wait y poner al thread en espera. Como esta verificación puede ocurrir varias veces cuando se intenta ejecutar el método sacar, lo ideal es ponerlo en un ciclo que verifique que la pila no este vacía. El código quedaría con el siguiente formato.

```
      public synchronized T sacar() throws ExcepcionPila {
            while (vec.size() == 0) {
                  try {
                        this.wait();
                  } catch (InterruptedException e) {
                        // ignore it...
                  }
            }
            if (indice == 0)
                  throw new ExcepcionPila("Pila vacía");
            T t = vec.get(indice);
            vec.remove(indice);
            indice--;
            return t;
      }
```


Nota: En el bloque de sacar, la llamada al método wait se realiza antes de que se extraigan elementos de la pila. Esto es porque la extracción no puede realizarse a menos que haya algún elemento disponible.

Como el ciclo en el que se encuentra el llamado a wait se encarga de verificar que la pila no se encuentre vacía, no es necesario el código que se encarga de verificar que el índice del elemento a extraer sea menor que cero, lo cual permite evitar lanzar una excepción cuando la pila esta vacía simplemente porque ese caso nunca va a ocurrir, si la pila esta vacía se entre en estado de espera.

Por otro lado, tampoco es necesario utilizar un índice porque el método remove de la clase ArrayList elimina el último elemento de la lista y retorna dicho elemento. Dada la suma de estos cambios, el código se modifica para quedar en el siguiente estado.

```
public synchronized T sacar() throws ExcepcionPila {
	T c;
	while (vec.size() == 0) {
		try {
			this.wait();
		} catch (InterruptedException e) {
			// ignorar
		}
	}
	c = vec.remove(vec.size() - 1);
	return c;
}
```

El método agregar utiliza notify() para liberar un thread colocado en el grupo de espera del objeto pila al invocar a wait. Una vez liberado el thread, éste puede obtener el bloqueo sobre la memoria compartido del objeto del tipo Pila que se esté utilizando para que este pueda ejecutar el método sacar, que extrae un carácter del ArrayList que es la memoria compartida de dicho objeto.

El método agregar

Al igual que el método sacar, accede a la memoria compartida, motivo por el cual el bloque lo define debe estar sincronizado. La primera modificación al código también resulta sencilla en este caso.

```
public synchronized void agregar(T valor){
	vec.add(valor);
	indice++;
}
```

Dado que agregar coloca un elemento en la pila, es responsable de notificar a los threads en espera que la pila ya no está vacía. Esta notificación se realiza para los threads que se encuentran en espera para el mismo objeto sobre el cual se realiza la notificación. Es importante tener en cuenta que dado que el bloqueo se realiza sobre las variables de instancia, los estados de espera y notificación siempre actúan sobre un mismo objeto y si hubiese más de un objeto del tipo pila, se producirían estados de espera y notificación para cada uno de ellos diferenciándolos por instancia.

La llamada a notify sirve para liberar un único hilo que ha llamado a wait porque la pila estaba vacía. Realizar la llamada a notify antes de que cambien los datos compartidos no tiene ninguna consecuencia porque la pila seguiría vacía si la intentase acceder un thread que invoque al método sacar y lo pondría de nuevo en espera, pero principalmente porque el indicador de bloqueo no se libera hasta que se haya terminado el bloque sincronizado que lo obtuvo, que es el lugar donde se ejecuta notify. Agregando la notificación el método agregar queda de la siguiente manera.

```
public synchronized void agregar(T valor) {
	vec.add(valor);
```

```
            indice++;
            notify();
      }
```

A primera vista puede parecer que es mucho más fácil agregar elementos que sacarlos, pero la verdadera razón es que el código sólo verifica que la pila no esté vacía, lo cual afecta sólo al método sacar. Si se realizara una verificación del tamaño máximo del ArrayList, el método agregar sería un poco más complicado.

Versión final de la clase Pila

Con la modificación de ambos métodos incluida, el código modificado para la clase Pila queda como se muestra a continuación.

Ejemplo

```
package pila;
import java.util.ArrayList;

public class Pila <T>{
      private ArrayList<T> vec = new ArrayList<T>();
      private int indice = 0;

      public synchronized void agregar(T valor){
            vec.add(valor);
            indice++;
      }

      public synchronized T sacar() throws ExcepcionPila {
            if (indice == 0)
                  throw new ExcepcionPila("Pila vacía");
            T t = vec.get(indice);
            vec.remove(indice);
            indice--;
            return t;
      }
}
```

Productor

A continuación se busca crear una clase que pueda producir elementos para un objeto de tipo de la clase Pila. La clase a crear debe poder llamar al método agregar del objeto del tipo Pila en un thread independiente porque se desea que un programa pueda crear tantos productores como le sea necesario.

Ejemplo

```
package pila;

public class Productor implements Runnable {
        private Pila<Character> pila;
        private int nroThread;
```

```
        private static int contador = 1;

        public Productor (Pila<Character> s) {
          pila = s;
          nroThread = contador++;
        }

      public void run() {
                char c;

                 for (int i = 0; i < 200; i++) {
                   c = (char)(Math.random() * 26 +'A');
                   pila.agregar(c);
                   System.out.println("Productor" + nroThread + ": " + c);
                   try {
                     Thread.sleep((int)(Math.random() * 300));
                   } catch (InterruptedException e) {
                     // ignorar
                   }
                 }
      }
}
```

Consumidor

Análogamente, se desea crear una clase que pueda consumir elementos de un objeto del tipo Pila. La clase a crear debe poder llamar al método sacar del objeto del tipo Pila en un thread independiente porque se desea que un programa pueda crear tantos consumidores como le sea necesario.

Ejemplo

```
package pila;

public class Consumidor implements Runnable {
        private Pila<Character> pila;
        private int nroThread;
        private static int contador = 1;

        public Consumidor(Pila<Character> s) {
          pila = s;
          nroThread = contador++;
        }

      public void run() {
          char c = 0;
          for (int i = 0; i < 200; i++) {
            try {
                   c = pila.sacar();
             } catch (ExcepcionPila e1) {
                   System.out.println("Error: " + e1.getMessage());
                   e1.printStackTrace();
             }
            System.out.println("Consumidor" + nroThread + ": " + c);
```

```
            try {
              Thread.sleep((int)(Math.random() * 300));
            } catch (InterruptedException e) {
              // ignorar
            }
          }
      }
}
```

La clase VerificarPilaSincronizada

El último paso restante para probar como diferentes productores y distintos consumidores pueden trabajar sobre un mismo objeto del tipo pila es ejecutarlos en un programa. Utilizando caracteres como elementos de la pila, por una cuestión de simplicidad solamente, los productores producen caracteres y los consumidores los leen para presentarlos por pantalla en una salida por consola. Cabe notar que como la clase Pila es genérica, se puede utilizar cualquier tipo como elemento de ésta, pero las clases Productor y Consumidor que trabajan específicamente con los elementos que posea la pila, deberán ajustar su código en base al tipo que se quiera utilizar.

Ejemplo

```
package pila;

public class VerificarPilaSincronizada {

      public static void main(String[] args) {
          Pila<Character> pila = new Pila<Character>();

          Productor p1 = new Productor(pila);
          Thread prodT1 = new Thread (p1);
          prodT1.start();

          Productor p2 = new Productor(pila);
          Thread prodT2 = new Thread (p2);
          prodT2.start();

          Consumidor c1 = new Consumidor(pila);
          Thread consT1 = new Thread (c1);
          consT1.start();

          Consumidor c2 = new Consumidor(pila);
          Thread consT2 = new Thread (c2);
          consT2.start();
      }
}
```

Una muestra ***acotada*** de la posible salida (recordar que los caracteres se generan al azar) es la siguiente:

```
Productor2: K
Productor1: Z
Consumidor2: K
```

```
Consumidor1: Z
Productor1: V
Consumidor2: V
Productor2: C
Consumidor1: C
Productor1: R
Consumidor1: R
Productor2: E
Consumidor2: E
Productor1: F
Consumidor1: F
Productor2: K
Consumidor1: K
:

:
```