

PROGRAMACIÓN MULTIHILO EN JAVA



Clase 10:

**Utilerías de concurrencia
(Parte I)**

El API para concurrencia

- El API de concurrencia está contenido en el paquete **java.util.concurrent** y en sus dos subpaquetes **java.util.concurrent.atomic** y **java.util.concurrent.locks**.
- **java.util.concurrent**: Define las características principales de sincronización y comunicación entre hilos. Define las siguientes características:
 - ❖ Sincronizadores
 - ❖ Ejecutores
 - ❖ Colecciones concurrentes
- **java.util.concurrent.atomic**: Provee una forma eficiente de actualizar el valor de una variable sin el uso de candados. Esto se logra a través del uso de clases, tales como **AtomicInteger** y **AtomicLong**, y métodos como, **compareAndSet()**, **decrementAndGet()** y **getAndSet()**.
- **java.util.concurrent.locks**: Proporciona una alternativa al uso de métodos de sincronizados. El núcleo de esta alternativa es la interfaz **Lock**, la cual define el mecanismo básico utilizado para adquirir y liberar el acceso a un objeto. Los métodos clave son **lock()**, **tryLock()** y **unlock()**. La ventaja de utilizar estos métodos es el mayor control sobre la sincronización.

Sincronizadores

- Los sincronizadores ofrecen formas de sincronización de alto nivel para la interacción entre múltiples hilos. Las clases sincronizadoras definidas por **java.util.concurrent** son:

Semaphore	Implementa al semáforo clásico.
CountDownLatch	Espera hasta que un número especificado de eventos hayan ocurrido.
CyclicBarrier	Habilita a un grupo de hilos para esperar en un punto predefinido de ejecución.
Exchanger	Intercambia datos entre dos hilos.

Semaphore

- La clase **Semaphore** tiene los dos constructores:
 - ❖ Semaphore(int num)
 - ❖ Semaphore(int num, boolean p)
- Donde, *num* especifica el valor inicial del contador. Esto es, *num* especifica el número de hilos que pueden acceder a un recurso compartido al mismo tiempo.
- Para utilizar un semáforo, el hilo que quiere acceder al recurso compartido intenta adquirir un permiso. Si el contador del semáforo es mayor que cero, entonces el hilo adquiere el permiso lo cual causa que el contador sea decrementado. Cuando el hilo no necesita más el acceso al recurso compartido, libera el permiso, lo cual causa que el contador sea incrementado.
- Por omisión, se asigna el permiso a los hilos que esperan por él en un orden indefinido.
- Establecer el valor *p* a **verdadero**, asegura que el permiso sea asignado en el orden en el que los hilos solicitaron el acceso.

- Para adquirir el permiso, se llama al método **acquire()**, el cual tiene dos formas:
 - ❖ `void acquire()`
 - ❖ `void acquire(int num)`
- La primera forma adquiere un permiso. La segunda forma adquiere el número de permisos definidos en *num*.
- Si el permiso no puede ser obtenido al tiempo de la llamada, entonces el hilo que invoca se suspende hasta que el permiso le sea otorgado.
- Para liberar el permiso, se llama al método **release()**, el cual tiene dos formas:
 - ❖ `void release()`
 - ❖ `void release(int num)`
- La primera forma libera un permiso. La segunda forma libera el número de permisos especificados en *num*.
- Para utilizar un semáforo como control de acceso a un recurso, cada hilo que quiere usar el recurso debe primero llamar al método **acquire()** antes de acceder al recurso. Cuando el hilo ha terminado de utilizar el recurso, debe llamar al método **release()**.

Ejemplo 1

- En este ejemplo se comparte una variable **contador** que incrementa o decrementa de forma sincronizada, es decir que mientras esta haciendo una operación (incremento o decremento) la otra tiene que esperar.

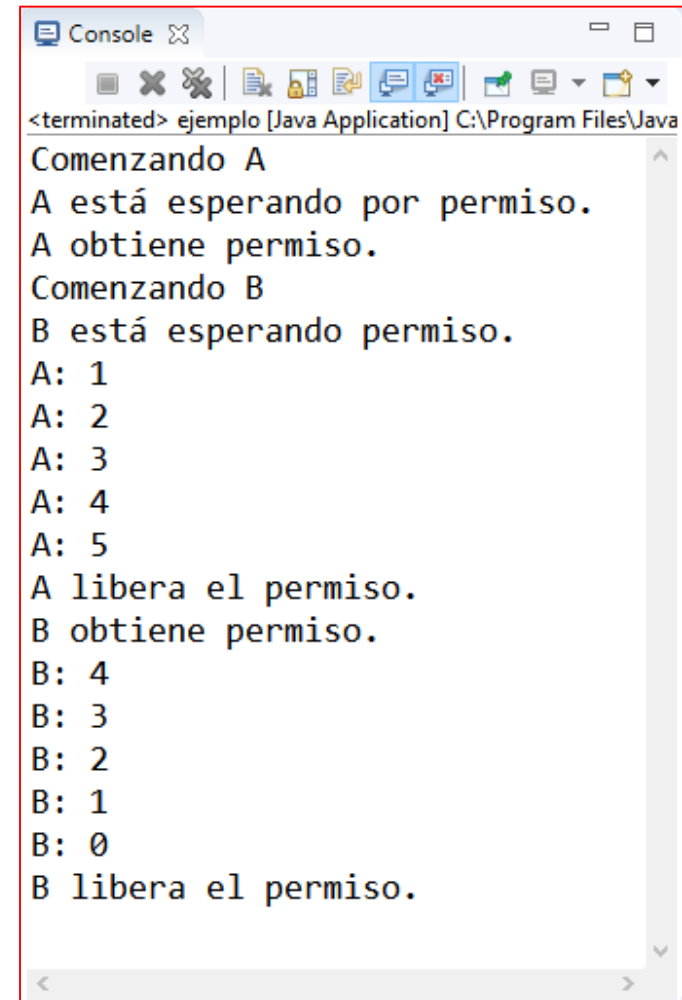
//Un ejemplo simple de objetos de tipo Semaphore

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

class ejemplo {
    public static void main(String args[]) {
        Semaphore sem = new Semaphore(1);
        new HiloIncrementa(sem, "A");
        new HiloDecrementa(sem, "B");
    }
}
```

//Un recurso compartido

```
class Compartido {
    static int contador = 0;
}
```



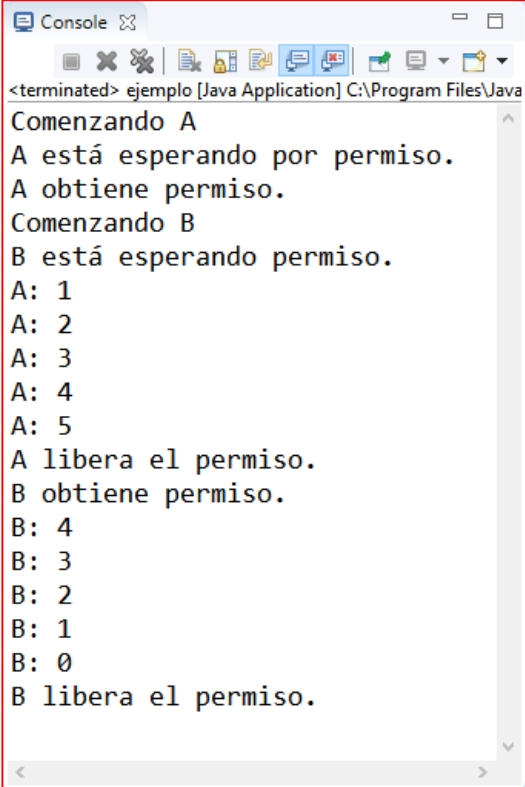
```
<terminated> ejemplo [Java Application] C:\Program Files\Java
Comenzando A
A está esperando por permiso.
A obtiene permiso.
Comenzando B
B está esperando permiso.
A: 1
A: 2
A: 3
A: 4
A: 5
A libera el permiso.
B obtiene permiso.
B: 4
B: 3
B: 2
B: 1
B: 0
B libera el permiso.
```

//Un hilo de ejecución que incrementa contador.

```
class HiloIncrementa implements Runnable {
    String name;
    Semaphore sem;
    HiloIncrementa(Semaphore s, String n) {
        sem = s;
        name = n;
        new Thread (this).start();
    }
    public void run() {
        System.out.println ("Comenzando " + name);
        try {
            // Primero, obtiene el permiso
            System.out.println(name + " está esperando por permiso.");
            sem.acquire();
            System.out.println(name + " obtiene permiso.");
            // Ahora, accede al recurso compartido
            for (int i=0; i < 5; i++) {
                Compartido.contador++;
                System.out.println(name + ": " + Compartido.contador);
                // Ahora, permite un cambio de contexto – si es posible.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Error: " + e.getMessage());
        }
        // Libera el permiso
        System.out.println (name + " libera el permiso.");
        sem.release();
    }
}
```

// Un hilo de ejecución que decrementa contador.

```
class HiloDecrementa implements Runnable {
    String name;
    Semaphore sem;
    HiloDecrementa (Semaphore s, String n) {
        sem = s;
        name = n;
        new Thread(this).start();
    }
    public void run () {
        System.out.println("Comenzando " + name);
        try {
            // Primero obtiene el permiso.
            System.out.println(name + " está esperando permiso.");
            sem.acquire ();
            System.out.println(name + " obtiene permiso.");
            // Ahora, accede al recurso compartido
            for (int i=0; i < 5; i++) {
                Compartido.contador--;
                System.out.println (name + ": " + Compartido.contador);
                //Ahora, permite un cambio de contexto – si es posible
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Error: " + e.getMessage());
        }
        //Libera el permiso
        System.out.println(name + " libera el permiso.");
        sem.release();
    }
}
```



```
<terminated> ejemplo [Java Application] C:\Program Files\Java
Comenzando A
A está esperando por permiso.
A obtiene permiso.
Comenzando B
B está esperando permiso.
A: 1
A: 2
A: 3
A: 4
A: 5
A libera el permiso.
B obtiene permiso.
B: 4
B: 3
B: 2
B: 1
B: 0
B libera el permiso.
```


Ejemplo 2

- Programa del productor/consumidor utilizando dos semáforos para regular a los hilos productor y consumidor, asegurando que cada llamada al método **poner()** es seguida por la correspondiente llamada al método **recoger()**.

//Una implementación de un productor y consumidor

//que utiliza semáforos para controlar la sincronización

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

```
class Monitor {
```

```
    int n;
```

```
    //Comienza con el semáforo no disponible para el consumidor
```

```
    static Semaphore semCon = new Semaphore(0);
```

```
    static Semaphore semProd = new Semaphore(1);
```

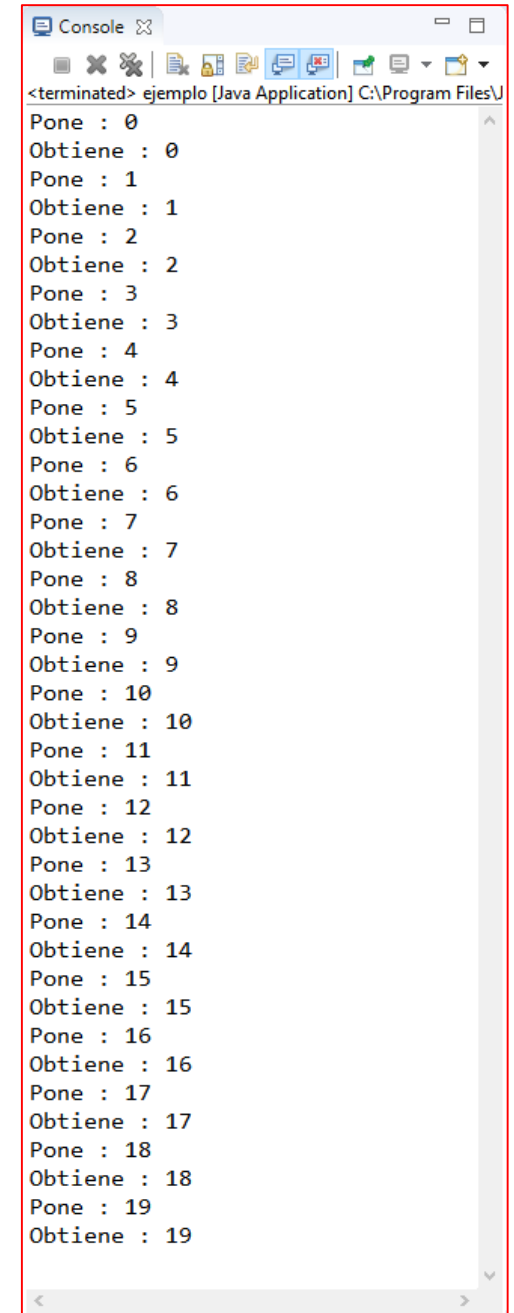
```
void recoger() {  
    try {  
        semCon.acquire();  
    } catch (InterruptedException e) {  
        System.out.println("Error: " + e.getMessage());  
    }  
    System.out.println("Obtiene : " + n);  
    semProd.release ();  
}  
void poner(int n) {  
    try {  
        semProd.acquire();  
    } catch ( InterruptedException e) {  
        System.out.println("Error: " + e.getMessage());  
    }  
  
    this.n = n;  
    System.out.println("Pone : " + n);  
    semCon.release ();  
}
```

```
}
```

```
class Productor implements Runnable {  
    Monitor q;  
    Productor(Monitor q) {  
        this.q = q;  
        new Thread(this, "Productor").start();  
    }  
    public void run() {  
        for(int i=0; i < 20; i++) q.poner(i);  
    }  
}
```

```
class Consumidor implements Runnable {  
    Monitor q;  
    Consumidor(Monitor q) {  
        this.q = q;  
        new Thread(this, "Consumidor").start();  
    }  
    public void run(){  
        for(int i=0; i < 20; i++) q.recoger();  
    }  
}
```

```
class ejemplo {  
    public static void main(String args[]) {  
        Monitor q = new Monitor();  
        new Consumidor(q);  
        new Productor(q);  
    }  
}
```



```
Console  
<terminated> ejemplo [Java Application] C:\Program Files\J  
Pone : 0  
Obtiene : 0  
Pone : 1  
Obtiene : 1  
Pone : 2  
Obtiene : 2  
Pone : 3  
Obtiene : 3  
Pone : 4  
Obtiene : 4  
Pone : 5  
Obtiene : 5  
Pone : 6  
Obtiene : 6  
Pone : 7  
Obtiene : 7  
Pone : 8  
Obtiene : 8  
Pone : 9  
Obtiene : 9  
Pone : 10  
Obtiene : 10  
Pone : 11  
Obtiene : 11  
Pone : 12  
Obtiene : 12  
Pone : 13  
Obtiene : 13  
Pone : 14  
Obtiene : 14  
Pone : 15  
Obtiene : 15  
Pone : 16  
Obtiene : 16  
Pone : 17  
Obtiene : 17  
Pone : 18  
Obtiene : 18  
Pone : 19  
Obtiene : 19
```

CountDownLatch

- Algunas veces se desea que un hilo espere hasta que uno o más eventos hayan ocurrido.
- Para manejar este tipo de situaciones, el API de concurrencia proporciona al **CountDownLatch**.
- Un **CountDownLatch** es creado inicialmente con un contador del número de eventos que deben ocurrir antes de que el bloqueo se libere. Cada vez que un evento ocurre, el contador es decrementado. Cuando el contador alcanza cero, el bloqueo se libera.
- **CountDownLatch** tiene el siguiente constructor `CountDownLatch (int num)`
- Donde, *num* especifica el número de eventos que deben ocurrir para que el bloqueo se libere.
- Para esperar a un bloqueo, un hilo llama al método **await()**, el cual tiene la forma:
 - ❖ `void await() throws InterruptedException`
 - ❖ `void await(long esp, TimeUnit tu) throws InterruptedException`
- La primera forma espera hasta que el contador asociado con el **CountDownLatch** alcance a cero. En la segunda forma espera solo por el periodo de tiempo especificado en el argumento llamado *esp*. Las unidades representadas por *esp* son especificadas por *tu*, el cual es un objeto de tipo enumeración **TimeUnit**.
- Una señal de un evento, llama al método **countDown()**, como se muestra aquí:
 - ❖ `void countDown()`
- Cada llamada a **countDown()** decrementa al contador asociado con el objeto que invoca.

Ejemplo 3

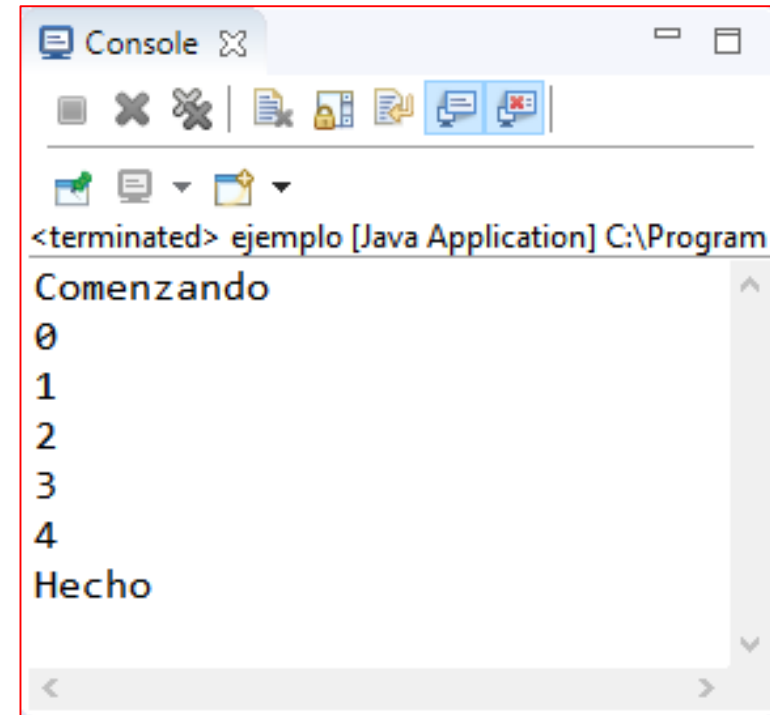
- Cree un bloqueo que requiere que ocurran cinco eventos antes de que se libere.

//Un ejemplo de CountdownLatch

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

class ejemplo {
    public static void main(String args[]) {
        CountdownLatch cdl = new CountdownLatch(5);
        System.out.println("Comenzando");
        new MiHilo(cdl);
        try {
            cdl.await();
        } catch (InterruptedException e) {
            System.out.println("Error: " + e.getMessage());
        }
        System.out.println("Hecho");
    }
}
```

```
class MiHilo implements Runnable {  
    CountdownLatch latch;  
    MiHilo(CountDownLatch c) {  
        latch = c;  
        new Thread(this).start();  
    }  
    public void run() {  
        for(int i = 0 ; i<5; i++) {  
            System.out.println (i);  
            latch.countDown(); // decrementa el contador  
        }  
    }  
}
```



The screenshot shows a console window titled "Console" with a toolbar containing icons for running, debugging, and other IDE functions. The console output is as follows:

```
<terminated> ejemplo [Java Application] C:\Program  
Comenzando  
0  
1  
2  
3  
4  
Hecho
```

CyclicBarrier

- Algunas veces se desea que un grupo de dos o más hilos esperen en un predeterminado punto de ejecución hasta que todos los hilos en el grupo hayan alcanzado ese punto.
- Para gestionar tal situación, el API de concurrencia provee la clase **CyclicBarrier**. La cual permite la definición de un objeto de sincronización que se suspende hasta que el número especificado de hilos haya alcanzado el punto establecido.
- **CyclicBarrier** tiene los siguientes dos constructores:
 - ❖ `CyclicBarrier(int numHilos)`
 - ❖ `CyclicBarrier(int numHilos, Runnable ac)`
- Donde, *numHilos* especifica el número de hilos que deben alcanzar la barrera antes de que la ejecución continúe. En la segunda forma, **ac** especifica a un hilo que será ejecutado cuando la barrera sea alcanzada.

- El procedimiento general que se debe seguir para utilizar a **CyclicBarrier**. Es el siguiente:
- Primero, se crea un objeto **CyclicBarrier**, especificando el número de hilos que se estará esperando.
- A continuación, cuando cada hilo alcance el límite, se llama al método **await()**. Esto detiene la ejecución del hilo hasta que todos los otros hilos también llamen al método **await()**.
- Una vez que el número especificado de hilos haya alcanzado el límite, **await()** regresará, y la ejecución continuará. Además, si se ha especificado una acción en el parámetro, dicho hilo será ejecutado.
- El método **await()** tiene las siguientes dos formas:
 - ❖ `int await() throws InterruptedException, BrokenBarrierException`
 - ❖ `int await(long esp, TimeUnit tu) throws InterruptedException, BrokenBarrierException, TimeoutException`
- La primera forma espera hasta que todos los hilos hayan alcanzado el punto límite. La segunda forma espera sólo por el periodo de tiempo especificado por *esp*. Las unidades representadas por *esp* están especificadas por *tu*. Ambas formas regresan un valor que indica el orden en que los hilos llegan al punto límite. El primer hilo devuelve un valor igual al número de hilos a ser atendidos menos uno. El último hilo devuelve cero.

Ejemplo 4

- Este programa espera hasta que un grupo de tres hilos han alcanzado el límite. Cuando eso ocurre se ejecuta otro hilo.

//Ejemplo de CyclicBarrier.

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

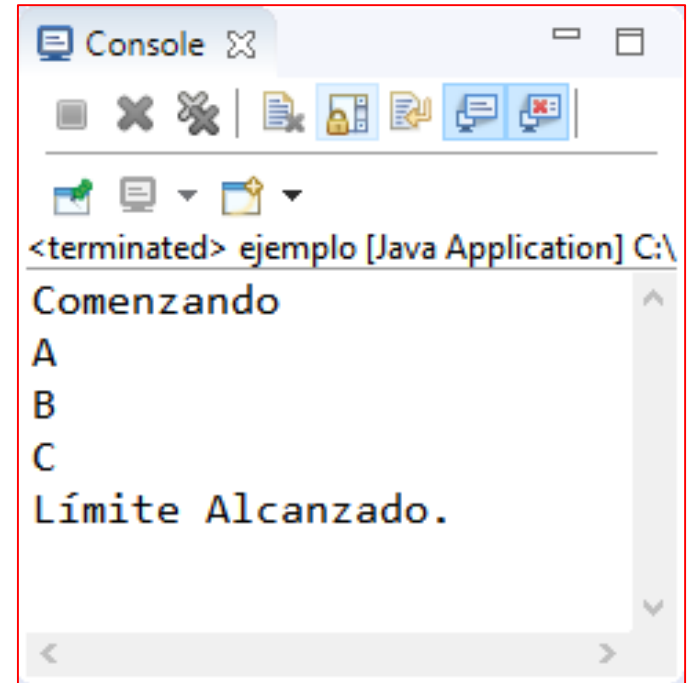
class ejemplo {
    public static void main(String args[]) {
        CyclicBarrier cb = new CyclicBarrier(3, new BarAccion());
        System.out.println("Comenzando");
        new MiHilo(cb, "A");
        new MiHilo(cb, "B");
        new MiHilo(cb, "C");
    }
}
```

//Un hilo de ejecución que utiliza un CyclicBarrier

```
class MiHilo implements Runnable {  
    CyclicBarrier cbar;  
    String nombre;  
    MiHilo (CyclicBarrier c, String n) {  
        cbar = c;  
        nombre = n;  
        new Thread(this).start();  
    }  
    public void run() {  
        System.out.println(nombre);  
        try {  
            cbar.await();  
        } catch (BrokenBarrierException e) {  
            System.out.println("Error: " + e.getMessage());  
        } catch (InterruptedException e) {  
            System.out.println("Error: " + e.getMessage());  
        }  
    }  
}
```

// Un objeto de esta clase es llamada cuando el CyclicBarrier termina.

```
class BarAccion implements Runnable {  
    public void run() {  
        System.out.println("Límite Alcanzado.");  
    }  
}
```



Exchanger

- Esta clase está diseñada para simplificar el intercambio de datos entre dos hilos.
- **Exchanger** espera hasta que dos hilos separados llamen al método **exchange()**. Cuando eso ocurre, se lleva a cabo el intercambio de datos proporcionados por los hilos.
- **Exchanger** es una clase genérica que se declara como:
 - ❖ `Exchanger <V>`
- Donde, **V** especifica el tipo de los datos que están siendo intercambiados.
- El único método definido por **Exchanger** es **exchange()**, el cual tiene las dos formas siguientes:
 - ❖ `V exchange(V bufer) throws InterruptedException`
 - ❖ `V exchange(V bufer, long esp, TimeUnit tu) throws InterruptedException, TimeoutException`
- Donde, *bufer* es una referencia a los datos a ser intercambiados. La segunda forma de **exchange()** permite especificar un periodo de tiempo de desconexión.
- El punto clave del método **exchange()** es que no terminará hasta que haya sido llamado para el mismo objeto **Exchanger** por dos hilos independientes. De esa forma, el método **exchange()** sincroniza el intercambio de datos.

Ejemplo 5

- El programa crea dos hilos. En un hilo se crea un buffer vacío que recibirá los datos que entregará el segundo hilo. De esta forma, el primer hilo intercambia un buffer vacío por uno lleno.

//Ejemplo de Exchanger.

```
import java.util.concurrent.Exchanger;
```

```
class ejemplo {
```

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

```
        Exchanger<String> exgr = new Exchanger<String> () ;
```

```
        new UsaCadena(exgr);
```

```
        new CreaCadena(exgr);
```

```
    }
```

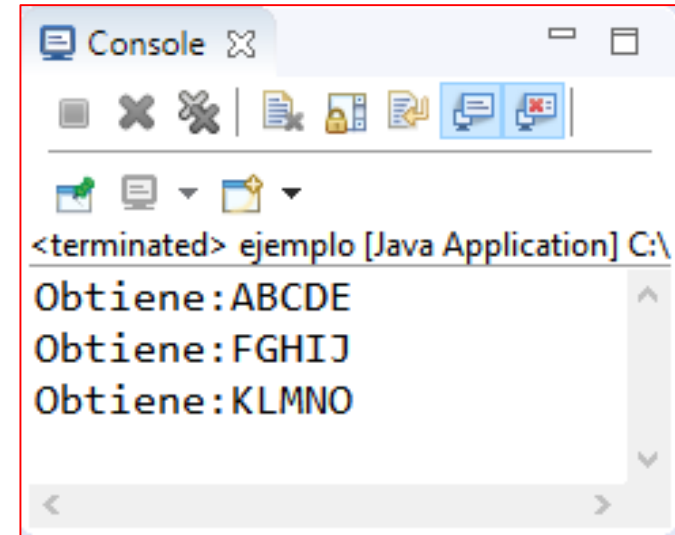
```
}
```

//Un hilo que construye una cadena.

```
class CreaCadena implements Runnable {  
    Exchanger<String> ex;  
    String str;  
    CreaCadena (Exchanger<String> c) {  
        ex = c;  
        str = new String();  
        new Thread (this).start();  
    }  
    public void run() {  
        char ch = 'A' ;  
        for (int i = 0; i < 3 ; i ++) {  
            //Llena el Bufer  
            for (int j = 0; j < 5; j++)  
                str += ch++;  
            try {  
                //Intercambia un bufer lleno por uno vacío.  
                str = ex. exchange(str);  
            } catch (InterruptedException e) {  
                System.out.println("Error: " + e.getMessage());  
            }  
        }  
    }  
}
```

//Un hilo que utiliza una cadena

```
class UsuCadena implements Runnable {  
    Exchanger<String> ex;  
    String str;  
    UsuCadena(Exchanger <String> c) {  
        ex = c;  
        new Thread(this).start();  
    }  
    public void run() {  
        for (int i = 0; i < 3; i++) {  
            try {  
  
                //Interchange a empty buffer for a full one  
                str = ex.exchange(new String());  
                System.out.println("Obtiene:" +str);  
            } catch (InterruptedException e) {  
                System.out.println("Error: " + e.getMessage());  
            }  
        }  
    }  
}
```



Ejercicio

- Crear 5 hilos.
- Cada hilo realiza una subsuma de 1 al 1000, 1001 al 2000, 2001 al 3000, etc.
- Espere que todos los hilos terminen, para realizar la suma.
- Muestre los suma total, resultados parciales y final

Solución

//Ejemplo de CyclicBarrier.

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

```
class ejemplo {
```

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

```
        CyclicBarrier cb = new CyclicBarrier(5, new BarAccion());
```

```
        System.out.println("Comenzando");
```

```
        new MiHilo(cb, "1", 1, 1, 1000);
```

```
        new MiHilo(cb, "2", 2, 1001, 2000);
```

```
        new MiHilo(cb, "3", 3, 2001, 3000);
```

```
        new MiHilo(cb, "4", 4, 3001, 4000);
```

```
        new MiHilo(cb, "5", 5, 4001, 5000);
```

```
    }
```

```
}
```

//Un hilo de ejecución que utiliza un CyclicBarrier

```
class MiHilo implements Runnable {
```

```
    CyclicBarrier cbar;
```

```
    String nombre;
```

```
    int x,y,k;
```

```
    static int suma[] = new int[6];
```

```
    MiHilo (CyclicBarrier c, String n, int k, int x, int y) {
```

```
        cbar = c;
```

```
        nombre = n;
```

```
        this.k=k;
```

```
        this.x=x;
```

```
        this.y=y;
```

```
        suma[k]=0;
```

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

```
    }
```

```

public void run() {
    for(int i=x;i<=y;i++)
        suma[k]=suma[k]+i;
    System.out.println("Hilo "+nombre+": "+suma[k]);
    try {
        cbar.await();
    } catch (BrokenBarrierException e) {
        System.out.println("Error: " + e.getMessage());
    } catch (InterruptedException e) {
        System.out.println("Error: " + e.getMessage());
    }
}
}

```

//Un objeto de esta clase es llamada cuando el CyclicBarrier termina.

```

class BarAccion implements Runnable {
    public void run() {
        int suma=0;
        for(int i=1;i<=5;i++)
            suma=suma+MiHilo.suma[i];
        System.out.println("Suma: "+suma);
    }
}

```