

MP_0490
Programación de
servicios y procesos



TEMA 1: Programas, Ejecutables, Procesos y Servicios

TEMA 1.2: Programación de aplicaciones multitarea (hilos)

Proceso vs hilo

Capturar el hilo principal

Crear un hilo extendiendo la clase Thread

Crear un hilo implementando la interfaz Runnable

Método join() sobre un hilo

Estados de un hilo

Resumen de librería y clases

Necesidad de sincronización

Cómo sincronizar hilos

En esta lección perseguimos los siguientes objetivos:

- 1 Comprender la diferencia entre multiproceso y multitarea.
- 2 Comprender el concepto de hilo de ejecución.
- 3 Realizar programas Java multitarea mediante el uso de varios hilos en ejecución.
- 4 Dominar el uso de la clase *Thread*.
- 5 Dominar el uso de la interfaz *Runnable*.

El hilo es la unidad mínima de procesamiento.

En la lección anterior, aprendiste que **un proceso es un programa en ejecución** y que pueden **lanzarse varios procesos simultáneos**. También vimos cómo los procesos son gestionados por el sistema operativo.



Pues bien, un **hilo** es la unidad mínima de procesamiento y se encuentra dentro de un proceso. Es decir, **un conjunto de instrucciones en ejecución dentro del contexto de un proceso**.

Todo proceso tendrá al menos un hilo en ejecución, aunque podría tener varios simultáneos, creando lo que se denomina **multitarea**.

Los hilos escapan al control del sistema operativo.

Esto nos lleva a distinguir entre los conceptos de multiproceso y multitarea.

1

Multiproceso: varios procesos que se ejecutan de manera concurrente y que son gestionados por el sistema operativo.

2

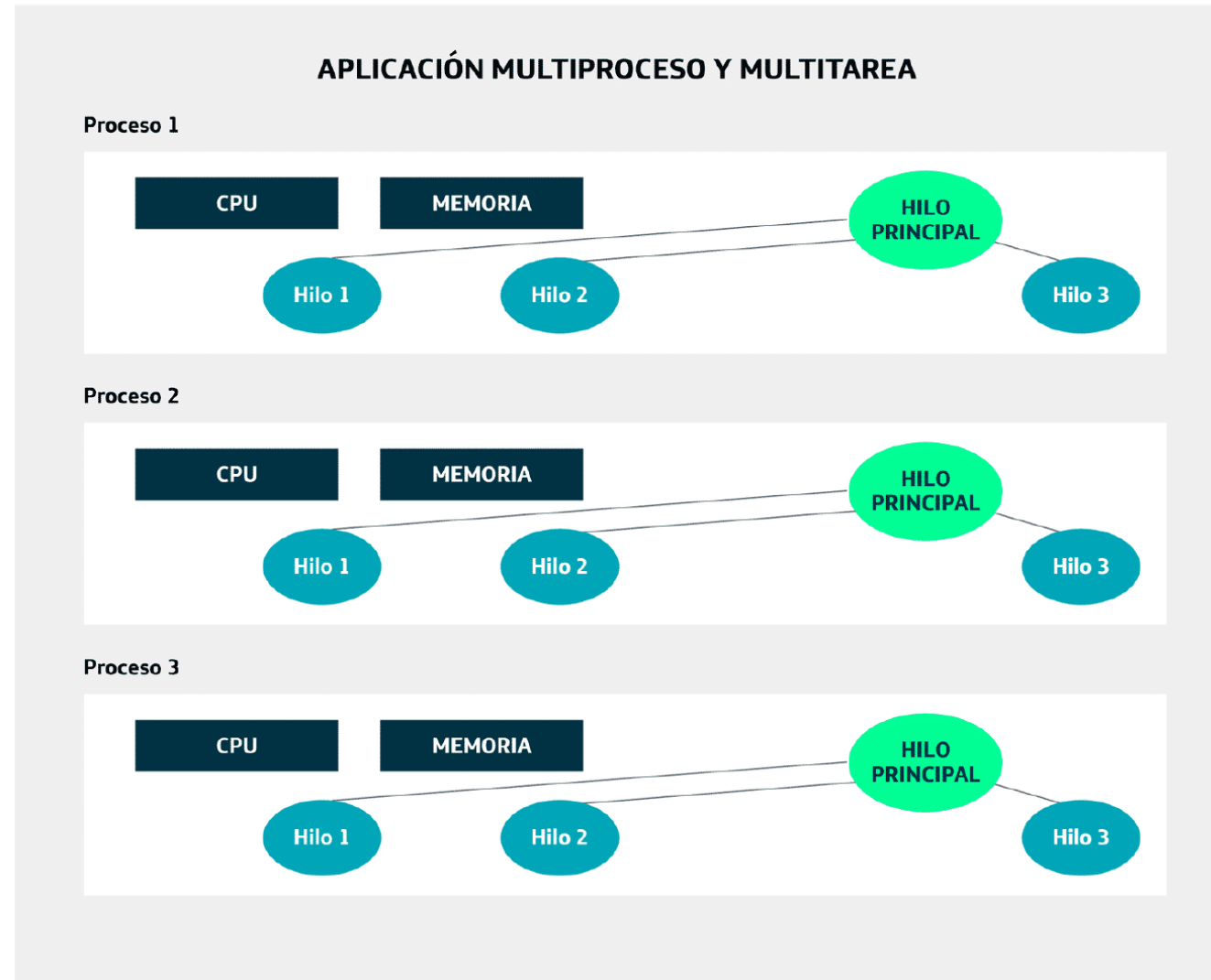
Multitarea: varios hilos de ejecución simultáneos dentro del mismo proceso.

Características de los programas multitarea

- Varios hilos en ejecución dentro del mismo proceso.
- Todos los hilos comparten los recursos que el sistema operativo haya asignado al proceso. La incorporación de un hilo más no supone la asignación de más recursos del sistema.
- Si el ordenador dispone de varios núcleos, cada hilo en ejecución puede aprovechar un núcleo distinto, produciéndose así la multitarea en el sentido estricto de la palabra.
- El uso de varios hilos de ejecución en un proceso es especialmente interesante en los programas en red de tipo cliente/servidor. Un hilo de ejecución puede estar atendiendo peticiones y, si el procesamiento de la respuesta es largo, podría generarse un hilo independiente para cada respuesta. De este modo, se podría estar atendiendo varias peticiones al mismo tiempo.

TEMA 1.2: Programación de aplicaciones multitarea (hilos)

Esquema de una aplicación multiproceso y multitarea



- ① Aunque todos los hilos lanzados desde el mismo proceso comparten la cuota de la CPU y la memoria asignada al proceso, **cada uno de ellos contará con un conjunto de registros de la CPU, un contador de programa y una pila para indicar por dónde se está ejecutando.**

Cada hilo en Java es un objeto de la clase *Thread*.

Podemos capturar el hilo principal de ejecución invocando al método estático *currentThread()* de la clase *Thread*. Cada hilo tiene un nombre que lo identifica y una prioridad, que pueden establecerse con los métodos *setName()* y *setPriority()*. El nombre por defecto es *main* y la prioridad 5.

En el siguiente programa, capturaremos el hilo principal y mostraremos su información con la llamada al método *toString()*. A continuación, haremos una pausa de 1 segundo, cambiaremos el nombre y prioridad del hilo, y volveremos a mostrar la información para comprobar cómo se han efectuado los cambios.

TEMA 1.2: Programación de aplicaciones multitarea (hilos)

```
Main.java ×  
1  ▶ public class Main  
2      {  
3  ▶      public static void main(String[] args) throws InterruptedException  
4          {  
5              System.out.println("Vamos a obtener información sobre hilo principal");  
6              Thread hilo = Thread.currentThread();  
7              System.out.println(hilo.toString());  
8              // Hacemos un retardo de un segundo.  
9              Thread.sleep( millis: 1000 );  
10             // Cambiamos nombre y prioridad.  
11             hilo.setPriority(3);  
12             hilo.setName("HiloPrincipal");  
13             System.out.println(hilo.toString());  
14         }  
15     }
```

```
"C:\Program Files\Java\jdk-23\bin\java.  
Vamos a obtener información sobre hilo  
Thread[#1,main,5,main]  
Thread[#1,HiloPrincipal,3,main]  
  
Process finished with exit code 0
```

En java, cuentas con dos sistemas para crear programas capaces de trabajar con dos o más hilos de ejecución simultanea:

- Crear una clase derivada de *Thread*.
- Crear un clase que implemente la interfaz *Runnable*.

Crear un hilo extendiendo la clase Thread

Vamos a poner en marcha la clase *Thread* con un programa sencillo que simula una carrera.

La clase *Corredor* representa a una persona que participa en la carrera.

```
public class Corredor extends Thread {  
  
    public Corredor(String nombre, int prioridad) {  
        super(nombre);  
        this.setPriority(prioridad);  
        this.start();  
    }  
  
    @Override  
    public void run() {  
        for (int i=1; i<=10; i++) {  
            System.out.println(this.getName() + " va por el kilómetro "  
+ i);  
        }  
        System.out.println(this.getName() + " ha llegado a la meta");  
    }  
}
```

La clase *Corredor* extiende de *Thread*, lo que le otorga la capacidad de multitarea. Será posible crear varios objetos de tipo *Corredor* que podrán correr simultáneamente. Cada objeto *Corredor* será un hilo de ejecución.

Establecer prioridades

En nuestro ejemplo, establecemos la prioridad de cada hilo por medio del segundo argumento al constructor, donde se invoca al método ***setPriority()*** con el valor de dicho argumento.

Vamos a otorgar la mayor prioridad al hilo con nombre *Corredor3* aunque, a pesar de eso, no podremos garantizar que llegará el primero; también tiene algo de ventaja el hilo que fue lanzado primero.

De hecho, si ejecutamos varias veces el programa, no siempre obtendremos la misma respuesta.

TEMA 1.2: Programación de aplicaciones multitarea (hilos)

```
public class Principal {  
    public static void main(String[] args) throws InterruptedException {  
        new Corredor("Corredor1",1);  
        new Corredor("Corredor2",3);  
        new Corredor("Corredor3",5);  
    }  
}
```

TEMA 1.2: Programación de aplicaciones multitarea (hilos)

El valor de la prioridad debe oscilar entre 1 y 10, valores que pueden ser consultados por las constantes estáticas *Thread.MIN_PRIORITY* y *Thread.MAX_PRIORITY*. Podríamos cambiar el código de la clase principal de la siguiente manera:

```
public class Principal {  
    public static void main(String[] args) throws InterruptedException {  
        new Corredor("Corredor1", Thread.MIN_PRIORITY);  
        new Corredor("Corredor2", Thread.MIN_PRIORITY);  
        new Corredor("Corredor3", Thread.MAX_PRIORITY);  
    }  
}
```

Ahora, los hilos *Corredor1* y *Corredor2* tendrán prioridad 1 y el hilo *Corredor3* tendrá prioridad 10.

¿Qué pasa si necesitamos que nuestra clase *Corredor* extienda, además, de otra segunda clase?

Podría darse la circunstancia de que tuviéramos implementada una clase *Persona* con las propiedades nombre, edad, peso, estatura, etc.

Puesto que un *Corredor* es una persona, nos podría venir bien hacer algo así:

```
public class Corredor extends Thread, Persona
{
    ....
}
```

la herencia múltiple no está permitida en Java. No sería correcto.

La solución a este tipo de problema está en crear hilos utilizando la interfaz *Runnable*

Crear un hilo implementando la interfaz Runnable

```
public class Corredor implements Runnable {
    private Thread hilo;

    public Corredor(String nombre, int prioridad) {
        hilo = new Thread(this, nombre);
        hilo.setPriority(prioridad);
        hilo.start();
    }

    @Override
    public void run() {
        for (int i=1; i<=10; i++) {
            System.out.println(hilo.getName() + " va por el kilómetro " + i);
        }
        System.out.println(hilo.getName() + " ha llegado a la meta");
    }
}
```

Puesto que estamos implementando una interfaz y no extendiendo una clase (herencia), todavía tenemos la posibilidad de extender una clase que nos interese.

IMPLEMENTAR HUMANO EN EL EJEMPLO

Método `join()` sobre un hilo

El método `join()` aplicado a un hilo, bloquea el subproceso de la función desde donde se invoca hasta que termina la ejecución del hilo sobre el que se aplica.

EJEMPLO

Queremos que el primer corredor realice su carrera en solitario y, una vez que haya terminado, competirán juntos el segundo y el tercer corredor. Para lograr que el primer corredor se adelante y los otros dos no empiecen hasta que este acabe, será necesario bloquear el resto de procesos hasta que termine su carrera.

TEMA 1.2: Programación de aplicaciones multitarea (hilos)


```
public class Corredor implements Runnable {
    private Thread hilo;

    public Corredor(String nombre, int prioridad) {
        hilo = new Thread(this, nombre);
        hilo.setPriority(prioridad);
        hilo.start();
    }

    public Thread getHilo() {
        return hilo;
    }

    @Override
    public void run() {
        for (int i=1; i<=10; i++) {
            System.out.println(hilo.getName() + " va por el kilómetro " + i);
        }
        System.out.println(hilo.getName() + " ha llegado a la meta");
    }
}
```

método *getHilo()* para tener acceso externo al objeto *Thread*

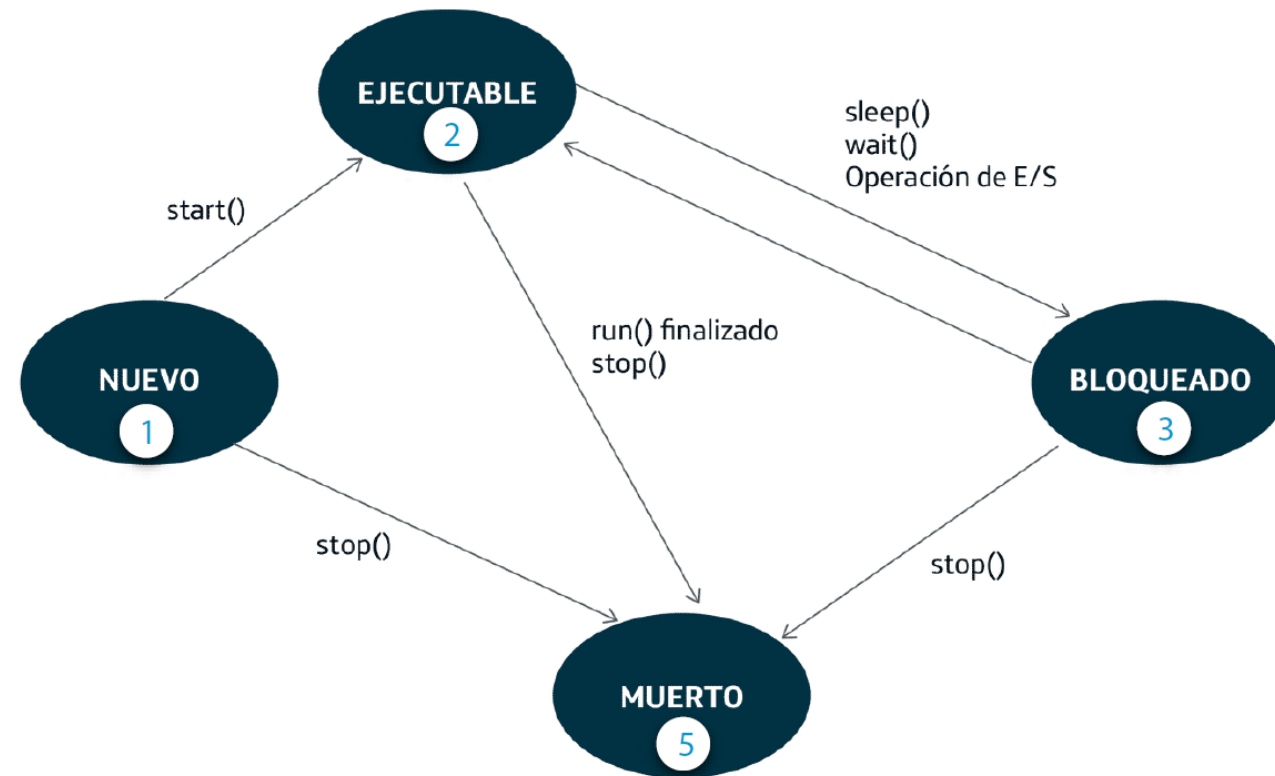


TEMA 1.2: Programación de aplicaciones multitarea (hilos)

```
public class Principal {  
    public static void main(String[] args) throws InterruptedException {  
        Corredor c1 = new Corredor("Corredor1",Thread.MIN_PRIORITY);  
        c1.getHilo().join();  
        new Corredor("Corredor2",Thread.MIN_PRIORITY);  
        new Corredor("Corredor3",Thread.MAX_PRIORITY);  
    }  
}
```

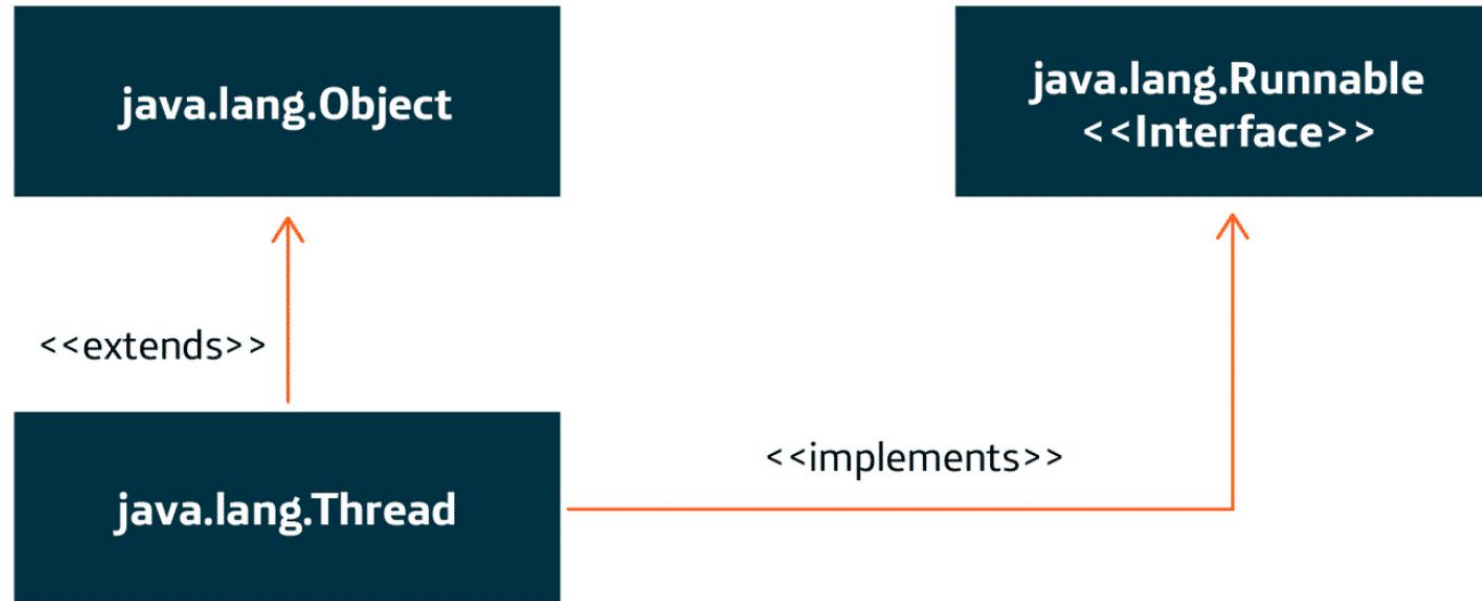
Estados de un hilo

Los hilos, igual que ocurre con los procesos, van cambiando de estado a lo largo de su ejecución.



Resumen de librería y clases

El diagrama para resumir las librerías y clases relacionadas con los hilos es muy simple, ya que se limita a la interfaz *Runnable* y la clase *Thread*.



La clase *Thread*, como todas las clases Java, extiende la superclase *Object*, de donde consigue métodos como *toString()*, *equals()*, *hashCode()*, etc. Por otro lado, también implementa la interfaz *Runnable*, de donde obtiene el método abstracto *run()*.

Métodos más importantes relacionados con los hilos

Thread currentThread(): devuelve la referencia al hilo que se encuentra en ejecución.

void start(): crea el hilo e inicia su ejecución, invocando al método *run()*.

void run(): ejecuta el método *run()*, que contendrá la lógica del hilo. No solemos invocar al método *run()* directamente, ya que es invocado por el método *start()*.

void sleep(int milisegundos): se suspende el hilo durante el número de milisegundos indicado.

void wait(): suspende el hilo hasta que sea despertado por otro hilo. Se utiliza en labores de sincronización. Es un método heredado de *Object*.

void wait(int milisegundos): igual que con *sleep()*, se suspende el hilo durante el número de milisegundos especificado, pero también puede ser despertado por otro hilo durante labores de sincronización. Es un método heredado de *Object*.

void interrupt(): interrumpe el hilo.

boolean interrupted(): devuelve *true* si el hilo ha sido interrumpido.

boolean isAlive(): devuelve *true* si el hilo sigue en ejecución, es decir, si no ha terminado su ejecución y llegado al estado de muerto.

void join(): bloquea el subproceso de la función desde donde se invoca hasta que termina la ejecución del hilo sobre el que se aplica el método *join()*.

A veces los distintos hilos de un proceso necesitan trabajar cooperativamente para alcanzar un objetivo común.

Varios hilos podrían compartir información accediendo a la misma variable, objeto, chero, etc. Esto podría crear **zonas críticas**, es decir, áreas de código que podrían crear problemas de concurrencia.

Vamos a crear una aplicación que utiliza dos clases, llamadas *Productor* y *Consumidor*.

Ambas clases implementan hilos que acceden a un objeto común, un objeto de la clase *BufferRadios*.

La clase *BufferRadios* contiene una *cola* (objeto *LinkedList*), que se llenará de números enteros que representan valores de los radios de varias circunferencias.

Los objetos de tipo *Productor* se encargarán de colocar elementos en la cola y los de tipo *Consumidor*, de sacar elementos de la cola y utilizarlos para calcular la longitud y el área de la circunferencia.

Explicar LinkedList

TEMA 1.2: Programación de aplicaciones multitarea (hilos)

```
import java.util.LinkedList;
import java.util.Queue;

public class BufferRadios {
    private Queue<Integer> cola;

    public BufferRadios() {
        cola = new LinkedList<Integer>();
    }

    public void poner(Integer r) {
        cola.add(r);
    }

    public Integer sacar() {
        if (cola.isEmpty()) {
            return null;
        }
        else {
            Integer radio = cola.remove();
            return radio;
        }
    }
}
```



Explicar LinkedList

La clase *Productor*

Implementa un hilo que actúa como productor de radios de circunferencias, generando números al azar y colocándolos en un objeto de tipo *BfferRadios*, llamado *buffer*, que será compartido por un consumidor.

FALTA PAGINA 22 DE CLASE

TEMA 1.2: Programación de aplicaciones multitarea (hilos)

Utilizamos la variable estática *contador* para establecer el nombre de cada hilo, como *Consumidor1*, *Consumidor2*, *Consumidor3*, ..., *ConsumidorN*.



```
public class Consumidor implements Runnable {
    private Thread hilo;
    private BufferRadios buffer;
    private static int contador = 0;

    public Consumidor(BufferRadios buffer) {
        contador++;
        hilo = new Thread(this, "Consumidor" + contador);
        this.buffer = buffer;
        hilo.start();
    }

    @Override
    public void run() {
        consumirUnRadio();
    }

    public void consumirUnRadio() {
        Integer radio = buffer.sacar();
        if (radio != null) {
            double l = 2 * Math.PI * radio;
            double a = Math.PI * radio * radio;
            System.out.println("Radio = " + radio + ", Longitud = " + l
+ ", Area = " + a);
        }
        else {
            System.out.println("Cola vacía");
        }
    }
}
```

TEMA 1.2: Programación de aplicaciones multitarea (hilos)

La clase *Principal* es la encargada de lanzar los hilos de ejecución, creando objetos de tipo *Productor* o *Consumidor*. Aquí se ve bien claro cómo los hilos productores y consumidores comparten el mismo objeto *buffer*.

```
public class Principal {  
    public static void main(String[] args) {  
        BufferRadios buffer = new BufferRadios();  
        new Productor(buffer);  
        new Consumidor(buffer);  
        new Productor(buffer);  
        new Consumidor(buffer);  
        new Productor(buffer);  
        new Consumidor(buffer);  
    }  
}
```


TEMA 1.2: Programación de aplicaciones multitarea (hilos)

Se supone que deseamos producir tres radios de circunferencia y consumir los tres. Sin embargo, el resultado de la ejecución es imprevisible. Ejecuta varias veces y comprueba los resultados.

```
Cola vacía
Productor2 va a generar un radio = 22
Productor1 va a generar un radio = 4
Cola vacía
Productor3 va a generar un radio = 26
Consumidor3 - Radio = 4, Longitud = 25.132741228718345, Area = 50.26548245743669
|
```

Se han producido tres radios, pero solo se ha podido consumir uno de ellos, el motivo es que los consumidores se han adelantado a los productores, intentando consumir antes de que los radios hayan sido colocados en el *buffer*.

```
Productor1 va a generar un radio = 12
Productor2 va a generar un radio = 29
Productor3 va a generar un radio = 22
Consumidor1 - Radio = 12, Longitud = 75.39822368615503, Area = 452.3893421169302
Consumidor2 - Radio = 29, Longitud = 182.212373908208, Area = 2642.079421669016
Consumidor3 - Radio = 22, Longitud = 138.23007675795088, Area = 1520.5308443374597
```

se han consumido todos los radios producidos

```
Productor1 va a generar un radio = 11
Cola vacía
Productor2 va a generar un radio = 14
Productor3 va a generar un radio = 23
Consumidor3 - Radio = 14, Longitud = 87.96459430051421, Area = 615.7521601035994
Consumidor2 - Radio = 11, Longitud = 69.11503837897544, Area = 380.1327110843649
```

Se ha perdido uno de los radios producidos.

Incluso si lo ejecutas muchas veces, terminará produciéndose una excepción

```
Productor1 va a generar un radio = 24
Cola vacía
Productor2 va a generar un radio = 28
Productor3 va a generar un radio = 13
Exception in thread "Consumidor3" java.util.NoSuchElementException
    at java.util.LinkedList.removeFirst(LinkedList.java:270)
    at java.util.LinkedList.remove(LinkedList.java:685)
    at BufferRadios.sacar(BufferRadios.java:20)
    at Consumidor.consumirUnRadio(Consumidor.java:19)
    at Consumidor.run(Consumidor.java:15)
    at java.lang.Thread.run(Thread.java:745)
Consumidor2 - Radio = 28, Longitud = 175.92918860102841, Area = 2463.0086404143976
```

La excepción la ha provocado el método *sacar()* de la clase *BufferRadios*, estamos intentando ejecutar un método *remove()* sobre una cola vacía.

TEMA 1.2: Programación de aplicaciones multitarea (hilos)

No lo teníamos controlado con la expresión condicional *cola.isEmpty()*?



```
public Integer sacar() {  
    if (cola.isEmpty()) {  
        return null;  
    }  
    else {  
        Integer radio = cola.remove();  
        return radio;  
    }  
}
```

Estamos trabajando con varios hilos de ejecución simultáneos. Seguro que la cola no estaba vacía en el momento de evaluarse el *if*, pero luego, antes de sacar el elemento con el método *remove()*, otro hilo se adelantó, colocándose en medio y sacando el último elemento que quedaba.

zona crítica!!

Cómo sincronizar hilos?

Métodos *wait()* y *notify()*

Los métodos *wait()* y *notify()* pertenecen a la superclase *Object*, por consiguiente, todos los objetos cuentan con ellos.

El método *wait()* deja bloqueado el hilo que lo llama, hasta que es liberado por otro hilo por medio de la ejecución del método *notify()* o cuando han transcurrido el número de milisegundos especificados. En nuestro ejemplo, mientras la cola esté vacía, bloquearemos repetitivamente la ejecución del método *sacar()* hasta que sea liberado por la ejecución del método *notify()*, llamado desde *poner()*, o hasta que transcurra un segundo.

El modificador *synchronized*

La utilización de los métodos *wait()* y *notify()* requiere de métodos marcados con el modificador *synchronized*.

El modificador *synchronized* se utiliza para indicar que un fragmento de código está **sincronizado**, es decir, que solamente un hilo puede acceder a dicho método a la vez.

Se podría armar que un método sincronizado tiene una marca de abierto y cerrado: cuando está cerrado ningún otro hilo puede entrar en dicho método.

TEMA 1.2: Programación de aplicaciones multitarea (hilos)

Actuaremos sobre la clase *BufferRadios*, que es la que contiene el método *sacar()* que constituye la zona crítica.

```
import java.util.LinkedList;
import java.util.Queue;

public class BufferRadios {
    private Queue<Integer> cola;

    public BufferRadios() {
        cola = new LinkedList<Integer>();
    }

    public synchronized void poner(Integer r) {
        cola.add(r);
        notify();
    }

    public synchronized Integer sacar() {
```

```
        while (cola.isEmpty()) {
            try {
                wait(1000);
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
        if (!cola.isEmpty()) {
            Integer radio = cola.remove();
            return radio;
        } else {
            return null;
        }
    }
}
```

Observa cómo el método *sacar()* ejecuta repetitivamente un *wait()* mientras la cola de radios está vacía.

Cada espera será de un segundo (1000 milisegundos) o menos.

Ahora mismo nuestro programa funciona perfectamente, pero

.....

si se construyeran solo dos objetos *Productor* y 3 objetos *Consumidor*, el último *Consumidor* entraría en un bucle infinito dentro de la estructura mientras está esperando a que el método *poner()* coloque un radio.

Esto podría solucionarse colocando un contador para limitar el número de repeticiones del *while*.

Vamos a cambiar el código para que solo se admita un máximo de veinte repeticiones.

TEMA 1.2: Programación de aplicaciones multitarea (hilos)

```
import java.util.LinkedList;
import java.util.Queue;

public class BufferRadios {
    private Queue<Integer> cola;

    public BufferRadios() {
        cola = new LinkedList<Integer>();
    }

    public synchronized void poner(Integer r) {
        cola.add(r);
        notify();
    }

    public synchronized Integer sacar() {
        int esperas = 0;
        while (cola.isEmpty() && esperas<20) {
            esperas ++;
            try {
                wait(10);
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
        if (esperas<20) {
            Integer radio = cola.remove();
            return radio;
        } else {
            return null;
        }
    }
}
```

TEMA 1.2: Programación de aplicaciones multitarea (hilos)

```
public class Consumidor implements Runnable {
    private Thread hilo;
    private BufferRadios buffer;
    private static int contador = 0;

    public Consumidor(BufferRadios buffer) {
        contador++;
        hilo = new Thread(this, "Consumidor" + contador);
        this.buffer = buffer;
        hilo.start();
    }

    @Override
    public void run() {
        consumirUnRadio();
    }

    public void consumirUnRadio() {
        Integer radio = buffer.sacar();
        if (radio != null) {
            double l = 2 * Math.PI * radio;
            double a = Math.PI * radio * radio;
            System.out.println(hilo.getName() + " - Radio = " + radio + ", Longitud
= " + l + ", Area = " + a);
        }
        else {
            System.out.println("Agotado tiempo de espera y no hay más elementos que
consumir");
        }
    }
}
```

Si transcurridas las veinte repeticiones todavía sigue vacía la lista, se dejará de esperar a que se produzcan más radios y nalizará el método *sacar()*, devolviendo el valor *null*.

¿Qué es el monitor?

La clave de la sincronización está en la palabra *monitor*. Como hemos comentado anteriormente, sólo un hilo de ejecución puede acceder a un método sincronizado al mismo tiempo;

se dice que ese hilo es el que **tiene el monitor y tendrá bloqueado el proceso hasta que finalice su ejecución y lo libere.**

MUCHAS GRACIAS POR VUESTRA ATENCIÓN!