# Tema 2.- Programación Multihilo.

## Objetivos

- Recursos compartidos por los hilos.
- Estados de un hilo. Cambios de estado.
- Elementos relacionados con la programación de hilos. Librerías y clases.
- Gestión de hilos.
- Sincronización de hilos.
- Compartición de información entre hilos.
- Programación de aplicaciones multihilo.

## Contenidos

### 1.- Introducción.

Existe una ley informática llamada ley de Wirth que dice:

***"El software se ralentiza más deprisa de lo que se acelera el hardware"***

Los programadores de aplicaciones, en la mayoría de los casos, no sabemos emplear la potencia del hardware y no se optimiza.

### 2.- Hilos en Java.

Un hilo es un objeto cuya clase hereda de **Thread** y el código que ejecuta el hilo debe introducirse en el método **run().** Mediante el método getName() del hilo actual currentThread() de la clase Thread obtenemos el nombre del hilo que se está ejecutando.

```
public class Hilo extends Thread {
     public void run() {
     ...
     }
}
```

Comienzan su ejecución con una llamada a **start**()

```
Hilo miHilo = new Hilo();
miHilo.start();
```

**Ejemplo1**: Crear un Hilo que muestra valores del 0 al 9.

```
public class Ejemplo1 {
    public static void main(String[] args) {
            Hilo miHilo = new Hilo();
            miHilo.start();
    }
}
```

```
class Hilo extends Thread {
    @Override
    public void run() {
    for (int i = 0; i < 10; i++)
        System.out.println("En el Hilo ... " + i);
    }
}
```

**Ejemplo2**: Crear tres Hilos que muestra valores del 0 al 9. Utilizamos el método getName() de del hilo actual Thread.currentThread() para mostrar el nombre del hilo que se está ejecutando.

```
public class Ejemplo2 {
    public static void main(String[] args) {
            Hilo miHilo0 = new Hilo();
            miHilo0.start();
            Hilo miHilo1 = new Hilo();
            miHilo1.start();
            Hilo miHilo2 = new Hilo();
            miHilo2.start();
    }
}
class Hilo extends Thread {
    @Override
    public void run() {
    for (int i = 0; i < 10; i++)
        System.out.println("En  el  Hilo  "+Thread.currentThread().getName()+
          "... " + i);
    }
}
```

**Ejecución:**

```
run:
En el Hilo Thread-2... 0
En el Hilo Thread-2... 1
En el Hilo Thread-1... 0
En el Hilo Thread-1... 1
En el Hilo Thread-1... 2
En el Hilo Thread-1... 3
En el Hilo Thread-1... 4
En el Hilo Thread-1... 5
En el Hilo Thread-0... 0
En el Hilo Thread-1... 6
En el Hilo Thread-1... 7
En el Hilo Thread-2... 2
En el Hilo Thread-2... 3
En el Hilo Thread-2... 4
En el Hilo Thread-2... 5
En el Hilo Thread-2... 6
En el Hilo Thread-1... 8
En el Hilo Thread-1... 9
En el Hilo Thread-0... 1
En el Hilo Thread-2... 7
En el Hilo Thread-2... 8
En el Hilo Thread-2... 9
En el Hilo Thread-0... 2
En el Hilo Thread-0... 3
En el Hilo Thread-0... 4
En el Hilo Thread-0... 5
En el Hilo Thread-0... 6
En el Hilo Thread-0... 7
En el Hilo Thread-0... 8
En el Hilo Thread-0... 9
BUILD   SUCCESSFUL   (total
time: 0 seconds)
```

Otra posibilidad es crear Hilos mediante el uso de la Interface Runnable. Se utiliza cuando una clase ya deriva de otra y como no se permite la herencia múltiple en Java debemos utilizar interfaces.

La clase tiene que implementar la interface Runnable:

```
public class HiloInterface extends otraClase implements Runnable{
     public void run() {
          …..
     }
}
```

Pero la clase **no es del tipo Thread, es decir no es un hilo**, por lo que debemos crear una instancia del tipo Thread. Todas las operaciones sobre el hilo se harán con la instancia Thread.

```
HiloInterface hilo = new HiloInterface ();
Thread mihilo = new Thread(hilo);
hilo.start();
```

**Ejemplo3**: Crear hilo mediante la interface Runnable para clases que extienden de otras clases.

```
public class Ejemplo3 {
    public static void main(String[] args) {
        HiloInterface hilo = new HiloInterface();
        Thread miHilo = new Thread(hilo);
        miHilo.start();
    }
}
class HiloInterface implements Runnable{
    @Override
    public void run() {
        for (int i = 0; i < 10; i++)
            System.out.println("En el Hilo "+
                    Thread.currentThread().getName()+"... " + i);
        }
}
```

**Ejemplo4**: Crear tres hilos mediante la interface Runnable para clases que extienden de otras clases.

```
public class Ejemplo4 {
    public static void main(String[] args) {
        HiloInterface hilo = new HiloInterface();
        Thread miHilo0 = new Thread(hilo);
        miHilo0.start();
        Thread miHilo1 = new Thread(hilo);
        miHilo1.start();
        Thread miHilo2 = new Thread(hilo);
        miHilo2.start();
    }
}
class HiloInterface implements Runnable{
    @Override
    public void run() {
        for (int i = 0; i < 10; i++)
            System.out.println("En el Hilo "
                    +Thread.currentThread().getName()+"... " + i);
        }
}
```

**Ejemplo5**: Calcular la suma de los elementos de un array con y sin hilos para comprobar la diferencia de tiempo en obtener el resultado. El método join() espera a que los hilos terminen para continuar la ejecución del programa, mostrando el total y el tiempo de cálculo con hilos.

```
package ejemplo5;
import java.time.*;
import java.time.temporal.*;
import java.util.Arrays;

public class Ejemplo5 {
    public static void main(String[] args) {
        int suma=0;
        LocalTime horaInicio;
        LocalTime horaFin;
        long milisegundos;
        int[] vector = new int[100000000];

        for (int i=0; i<vector.length;i++){
            vector[i]=(int)(Math.random()*10);
        }
        //System.out.println(Arrays.toString(vector));

        // Sin Hilos
        horaInicio = LocalTime.now();
        int total=0;
        for (int i = 0; i < vector.length; i++){
            total +=  vector[i];
        }
        horaFin = LocalTime.now();
        milisegundos = horaInicio.until(horaFin, ChronoUnit.MILLIS);
        System.out.println("Total    sin    Hilos:    "+total+"    -    Tiempo:
"+milisegundos+" ms");


        // Con Hilos
        int cuarto = vector.length/4;
        Hilo hilo1 = new Hilo(0,cuarto-1,vector);
        Hilo hilo2 = new Hilo(cuarto,cuarto*2-1,vector);
        Hilo hilo3 = new Hilo(cuarto*2,cuarto*3-1,vector);
        Hilo hilo4 = new Hilo(cuarto*3,vector.length-1,vector);

        Thread miHilo1 = new Thread(hilo1);
        Thread miHilo2 = new Thread(hilo2);
        Thread miHilo3 = new Thread(hilo3);
        Thread miHilo4 = new Thread(hilo4);

        horaInicio = LocalTime.now();
        miHilo1.start();
        miHilo2.start();
        miHilo3.start();
        miHilo4.start();
```

```
        try {
            miHilo1.join();
            miHilo2.join();
            miHilo3.join();
            miHilo4.join();
        } catch (InterruptedException e){

        }

        suma = hilo1.getSuma()+hilo2.getSuma()+hilo3.getSuma()
                    +hilo4.getSuma();
        horaFin = LocalTime.now();
        milisegundos = horaInicio.until(horaFin, ChronoUnit.MILLIS);
        System.out.println("Total con Hilos: "+suma+" - Tiempo: "
                    +milisegundos+" ms");
    }
}

class Hilo implements Runnable {
    int inicio;
    int fin;
    int suma;
    int[] vector;

    public Hilo(int inicio, int fin, int[] vector){
        this.inicio = inicio;
        this.fin = fin;
        this.vector = vector;
    }

    public int getSuma(){
        return this.suma;
    }

    @Override
    public void run() {
        int total=0;
        for (int i = inicio; i <= fin; i++){
            total +=  vector[i];
        }
        this.suma=total;
    }
}
```

## 3.- Estados de un Hilo.

Un hilo puede estar en cuatro estados:

- **Inicial**: antes de ejecutar start(). En realidad aún no es un hilo.
- **En ejecución**: tras ejecutar start() y durante el método run().
- **Bloqueado**:

    - Por **sincronización** de hilos.
    - **Durmiendo**: **Thread.sleep**(milisegundos);
    - El hilo llama al método wait() y no volverá a ejecutarse hasta recibir mensajes notify() o notifyAll().
- **Finalizado**: Cuando el método run() finaliza o salta alguna excepción.

Podemos averiguar si el hilo está en ***ejecución* o b*loqueado*** con el método **isAlive**(). Además el método getState() devuelve el estado del método que puede ser:

- **NEW**: El hilo se ha instanciado pero no se ha arrancado (start).
- **RUNNABLE**: El hilo se está ejecutando.
- **BLOCKED**: El hilo está bloqeado.
- **WAITING**: El hilo esta esperando a otro hilo para llevar a cabo una determinada acción. Por ejemplo un hilo que llama al método wait() de un objeto, está esperando hasta que otro hilo llame al método notify() del objeto.
- **TIMED_WAITING**: Este estado es similar a WAITING solo que el hilo está esperando por un tiempo determinado, no está esperando a una notificación como es el caso anterior.
- **TERMINATED**: El hilo ha finalizado la ejecución de todo su contenido. Un hilo en este estado no puede volver a iniciarse.

## 4.- Finalización de un hilo.

El Programa acaba cuando finaliza el método **main()**. El Hilo acaba cuando finaliza el método run() normal o por una excepción no capturada. A veces es necesario **detener un hilo**, pero debido a cuestiones de concurrencia la detención debe ser **ordenada** y se realiza mediante el método **interrupt**().

El hilo debe controlar cuando se solicita su interrupción mediante el método **Thread**.**interrupted**(). Debemos tener en cuenta las siguientes consideraciones sobre el método **interrupted()**:

- El método interrupted() es de **clase** para poder llamarlo desde una clase que no herede de Thread. (Thread.interrupted())
- La llamada al método resetea un *flag de interrupción*. Alternativa:

```
Thread.currentThread().isInterrupted();
```

Por otro lado, el método **join()** espera a que los hilos terminen para continuar la ejecución del programa. Debe ir en un bloque try...catch para controlar la excepción InterruptedException.

## 5.- Riesgos y consejos.

Riesgos:

- Distintos flujos de ejecución accediendo a los mismos objetos del programa.
- Surgen todos los problemas de la programación concurrente.

Consejos:

- Programar hilos puede ser un perjuicio más que un beneficio.
- Hay que utilizar los hilos con sensatez.

## 6.-Gestión de Prioridades.

Los métodos setPriority() y getPriority() permiten asignar y recuperar la prioridad de los hilos. La prioridad es un valor entero entre 1 y 10, siendo 5 la prioridad normal NORMAL_PRIORITY, 1 la prioridad mínima MIN_PRIORITY y 10 la prioridad máxima MAX_PRIORITY. El planificador elige el hilo que debe ejecutarse en función de la prioridad asignada.

**Ejemplo6**: El método interrupt() detiene un hilo. Por otro lado isInterrupt() permite comprobar si el hilo está detenido. En el ejemplo un valor al azar determina si el hilo se detiene con una probabilidad del 30%.

```
public class Ejemplo6 {
    public static void main(String[] args) {
        Hilo miHilo0 = new Hilo();
        miHilo0.start();
        Hilo miHilo1 = new Hilo();
        miHilo1.start();
        Hilo miHilo2 = new Hilo();
        miHilo2.start();
    }
}
class Hilo extends Thread {
    @Override
    public void run() {
        double probabilidad;
        for(int i=0; i<10; i++){
            if(!isInterrupted()){
                System.out.println("En el Hilo "+
                   Thread.currentThread().getName()+"... " + i);
                probabilidad = Math.random();
                if (probabilidad<0.30){
                    System.out.println("Hilo "+
                          Thread.currentThread().getName()+" Detenido");
                    interrupt();
                }
            }
        }
    }
}
```

**Ejemplo6b:** Programa eh java que lance 10 procesos uno de ellos con prioridad máxima y comprobar que termina la ejecución el primero.

```
package ejemplo7;
public class Ejemplo7 {
    public static void main(String[] args) {
        lanzarHilos();
    }

    public static void lanzarHilos() {
        Hilo miHilo0 = new Hilo();
        Hilo miHilo1 = new Hilo();
        Hilo miHilo2 = new Hilo();
```

```
        Hilo miHilo3 = new Hilo();
        Hilo miHilo4 = new Hilo();
        Hilo miHilo5 = new Hilo();
        Hilo miHilo6 = new Hilo();
        Hilo miHilo7 = new Hilo();
        Hilo miHilo8 = new Hilo();
        Hilo miHilo9 = new Hilo();
        miHilo0.setPriority(Thread.NORM_PRIORITY + 5);
        miHilo1.setPriority(Thread.NORM_PRIORITY - 4);
        miHilo2.setPriority(Thread.NORM_PRIORITY - 4);
        miHilo3.setPriority(Thread.NORM_PRIORITY - 4);
        miHilo4.setPriority(Thread.NORM_PRIORITY - 4);
        miHilo5.setPriority(Thread.NORM_PRIORITY - 4);
        miHilo6.setPriority(Thread.NORM_PRIORITY - 4);
        miHilo7.setPriority(Thread.NORM_PRIORITY - 4);
        miHilo8.setPriority(Thread.NORM_PRIORITY - 4);
        miHilo9.setPriority(Thread.NORM_PRIORITY - 4);
        miHilo0.start();
        miHilo1.start();
        miHilo2.start();
        miHilo3.start();
        miHilo4.start();
        miHilo5.start();
        miHilo6.start();
        miHilo7.start();
        miHilo8.start();
        miHilo9.start();
    }
}

class Hilo extends Thread {

    @Override
    public void run() {
        int contador = 0;
        for (int i = 0; i < 100000; i++) {
            contador++;
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }

        }
        System.out.println(Thread.currentThread().getName()    +    ":"    +
contador);

    }
}
```

**Ejemplo7:** Programa eh java que mediante interface gráfica crea un hilo y mediante un botón de Parar podemos detenerlo.

```
//(2) Importar java.awt.event.*;
import java.awt.event.*;
import javax.swing.*;
public class Ejemplo7 {
    public static void main(String[] args) {
        Formulario miFormulario = new Formulario();
        miFormulario.setVisible(true);
    }
}

//(3) Implementar interface ActionListener
class Formulario extends JFrame implements ActionListener, Runnable {
    JPanel miCapa;
    JLabel hiloT;
    JLabel contadorHiloT;
    JButton interrumpirB;
    Thread miHilo;

    public Formulario(){
        this.setTitle("Mi Aplicación Java");
        this.setSize(500,300);
        this.setResizable(false);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);

        miCapa = new JPanel();
        miCapa.setLayout(null);

        hiloT = new JLabel();
        hiloT.setText("Hilo 0:");
        hiloT.setBounds(20, 30, 100, 30);
        //x, y, ancho, alto
        miCapa.add(hiloT);


        contadorHiloT = new JLabel();
        contadorHiloT.setBounds(120, 30, 100, 30);
        //x, y, ancho, alto
        miCapa.add(contadorHiloT);

        interrumpirB = new JButton();
        interrumpirB.setText("Parar");
        interrumpirB.setBounds(70, 70, 100, 30);
        //x, y, ancho, alto
        miCapa.add(interrumpirB);


        //(1) botón a la escucha
        interrumpirB.addActionListener(this);
```

```
        this.add(miCapa);

        miHilo = new Thread(this);
        miHilo.start();
    }

    //(4) Sobreescribir el método actionPerformed
    // y parametro e recoge información del evento
    // e.getSource devuelve quien originó el evento

        @Override
        public void actionPerformed(ActionEvent e){
            String nombre;
            if(e.getSource()==interrumpirB){
                miHilo.interrupt();
            }
        }

    @Override
       public void run() {
           boolean ejecutar = true;
           int i=0;
           while(ejecutar){
               if(!Thread.currentThread().isInterrupted()){
                   contadorHiloT.setText(Integer.toString(i));
                   i++;
                   try {
                       Thread.sleep(1000);
                       System.out.println(Thread.currentThread());
                   } catch (InterruptedException e){
                       Thread.currentThread().interrupt();
                   }
               }
           }
       }
}
```

**Ejemplo8:** Programa eh java que mediante interface gráfica crea tres hilos y mediante tres botones de Parar podemos detenerlos. Con los métodos `setPriority()` y `getPriority()` asignamos y recuperamos la prioridad de los hilos que hemos creado.

```
//(2) Importar java.awt.event.*;
import java.awt.event.*;
import javax.swing.*;
public class Ejemplo8 {
    public static void main(String[] args) {
        Formulario miFormulario = new Formulario();
        miFormulario.setVisible(true);
    }
}
```

```
//(3) Implementar interface ActionListener
class Formulario extends JFrame implements ActionListener, Runnable {
    JPanel miCapa;
    JLabel hilo0T;
    JLabel hilo1T;
    JLabel hilo2T;
    JLabel contadorHilo0T;
    JLabel contadorHilo1T;
    JLabel contadorHilo2T;

    JButton interrumpirHilo0B;
    JButton interrumpirHilo1B;
    JButton interrumpirHilo2B;
    Thread miHilo0;
    Thread miHilo1;
    Thread miHilo2;

    public Formulario(){
        this.setTitle("Mi Aplicación Java");
        this.setSize(150,300);
        this.setResizable(false);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);

        miCapa = new JPanel();
        miCapa.setLayout(null);

        hilo0T = new JLabel();
        hilo0T.setText("Hilo 0:");
        hilo0T.setBounds(20, 30, 100, 30);
        //x, y, ancho, alto
        miCapa.add(hilo0T);

        hilo1T = new JLabel();
        hilo1T.setText("Hilo 1:");
        hilo1T.setBounds(20, 60, 100, 30);
        //x, y, ancho, alto
        miCapa.add(hilo1T);

        hilo2T = new JLabel();
        hilo2T.setText("Hilo 2:");
        hilo2T.setBounds(20, 90, 100, 30);
        //x, y, ancho, alto
        miCapa.add(hilo2T);

        contadorHilo0T = new JLabel();
        contadorHilo0T.setBounds(70, 30, 100, 30);
        //x, y, ancho, alto
        miCapa.add(contadorHilo0T);

        contadorHilo1T = new JLabel();
        contadorHilo1T.setBounds(70, 60, 100, 30);
```

```
        //x, y, ancho, alto
        miCapa.add(contadorHilo1T);

        contadorHilo2T = new JLabel();
        contadorHilo2T.setBounds(70, 90, 100, 30);
        //x, y, ancho, alto
        miCapa.add(contadorHilo2T);


        interrumpirHilo0B = new JButton();
        interrumpirHilo0B.setText("Parar Hilo 0");
        interrumpirHilo0B.setBounds(20, 140, 100, 30);
        //x, y, ancho, alto
        miCapa.add(interrumpirHilo0B);

        interrumpirHilo1B = new JButton();
        interrumpirHilo1B.setText("Parar Hilo 1");
        interrumpirHilo1B.setBounds(20, 180, 100, 30);
        //x, y, ancho, alto
        miCapa.add(interrumpirHilo1B);

        interrumpirHilo2B = new JButton();
        interrumpirHilo2B.setText("Parar Hilo 2");
        interrumpirHilo2B.setBounds(20, 220, 100, 30);
        //x, y, ancho, alto
        miCapa.add(interrumpirHilo2B);

        //(1) botón a la escucha
        interrumpirHilo0B.addActionListener(this);
        interrumpirHilo1B.addActionListener(this);
        interrumpirHilo2B.addActionListener(this);

        this.add(miCapa);

        miHilo0 = new Thread(this);
        miHilo1 = new Thread(this);
        miHilo2 = new Thread(this);

        miHilo0.setPriority(Thread.NORM_PRIORITY+2);
        miHilo1.setPriority(Thread.NORM_PRIORITY);
        miHilo2.setPriority(Thread.NORM_PRIORITY-2);

        miHilo0.start();
        miHilo1.start();
        miHilo2.start();
    }

    //(4) Sobreescribir el método actionPerformed
    // y parametro e recoge información del evento
    // e.getSource devuelve quien originó el evento
        @Override
        public void actionPerformed(ActionEvent e){
```

```
            String nombre;
            if(e.getSource()==interrumpirHilo0B){
                miHilo0.interrupt();
            }
            if(e.getSource()==interrumpirHilo1B){
                miHilo1.interrupt();
            }
            if(e.getSource()==interrumpirHilo2B){
                miHilo2.interrupt();
            }
        }

    @Override
       public void run() {
           boolean ejecutar = true;
           int i=0;
           String nombreHilo;
           while(ejecutar){
               if(!Thread.currentThread().isInterrupted()){
                   nombreHilo = Thread.currentThread().getName();
                   System.out.print(nombreHilo);
                   System.out.println("                                    -
"+Thread.currentThread().getPriority());
                   switch (nombreHilo) {
                       case "Thread-1":
                           contadorHilo0T.setText(Integer.toString(i));
                           break;
                       case "Thread-2":
                           contadorHilo1T.setText(Integer.toString(i));
                           break;
                       case "Thread-3":
                           contadorHilo2T.setText(Integer.toString(i));
                           break;
                       default:
                           break;
                   }
                   i++;

               }
           }
       }
}
```

## 7.-Comunicación y Sincronización entre Hilos.

En ocasiones los hilos necesitan comunicarse unos con otros y lo realizan mediante un objeto compartido. Cuando en un programa tenemos varios hilos corriendo simultáneamente es posible que varios hilos intenten acceder a la vez a un mismo sitio (un fichero, una conexión, un array de datos) y es posible que la operación de uno de ellos entorpezca la del otro.

Para evitar estos problemas, hay que sincronizar los hilos.

**Ejemplo9:** Programa que ejecute dos hilos. El hiloA suma 300 al atributo c del objeto compartido contador y el hiloB resta 300 al atributo c del objeto contador.

```
public class Ejemplo9 {
    public static void main(String[] args) {
          Contador cont = new Contador(100);
          HiloA a = new HiloA("HiloA", cont);
          HiloB b = new HiloB("HiloB", cont);
          a.start();
          b.start();
    }
}
class Contador {
    private int c = 0;

    Contador(int c) {
            this.c = c;
    }
    public void incrementa() {
            c = c + 1;
    }
    public void decrementa() {
            c = c - 1;
    }
    public int getValor() {
            return c;
    }
}
class HiloA extends Thread {
    private Contador contador;
    public HiloA(String n, Contador c) {
            setName(n);
            contador = c;
    }
    public void run() {
        for (int j = 0; j < 300; j++) {
            contador.incrementa();
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) { }
        }
        System.out.println(getName() + " contador vale "
          + contador.getValor());
    }
}

class HiloB extends Thread {
    private Contador contador;
    public HiloB(String n, Contador c) {
            setName(n);
            contador = c;
```

```
    }
    public void run() {
        for (int j = 0; j < 300; j++) {
            contador.decrementa();
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) { }
        }
        System.out.println(getName() + " contador vale "
          + contador.getValor());
    }
}
```

La ejecución podría parecer que devuelve HiloA 400 e HiloB 100 pero no es así pues los dos hilos acceden a la vez al objeto compartido pudiendo variar el resultado final.

```
run:
HiloB contador vale 114
HiloA contador vale 114
BUILD SUCCESSFUL (total time: 4 seconds)
```

Para que un bloque de código se ejecute de forma atómica, en la zona crítica de código utilizaremos la sentencia synchronized y entre paréntesis la referencia al objeto compartido.

```
public class Ejemplo9 {
    public static void main(String[] args) {
           Contador cont = new Contador(100);
           HiloA a = new HiloA("HiloA", cont);
           HiloB b = new HiloB("HiloB", cont);
           a.start();
           b.start();
    }
}

class Contador {
    private int c = 0;

    Contador(int c) {
            this.c = c;
    }
    public void incrementa() {
            c = c + 1;
    }
    public void decrementa() {
            c = c - 1;
    }
    public int getValor() {
            return c;
    }
}
```

```
class HiloA extends Thread {
    private Contador contador;
    public HiloA(String n, Contador c) {
            setName(n);
            contador = c;
    }
    public void run() {
        synchronized(contador){
            for (int j = 0; j < 300; j++) {
                contador.incrementa();
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) { }
            }
            System.out.println(getName()   +   "   contador   vale   "   +
contador.getValor());
        }
    }
}

class HiloB extends Thread {
    private Contador contador;
    public HiloB(String n, Contador c) {
            setName(n);
            contador = c;
    }
    public void run() {
        synchronized(contador){
            for (int j = 0; j < 300; j++) {
                contador.decrementa();
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) { }
            }
            System.out.println(getName()   +   "   contador   vale   "   +
contador.getValor());
        }
    }
}
```

La ejecución devuelve HiloA 400 e HiloB 100 ya que primero se ejecuta de forma atómica el hilo A y después el hilo B, accediendo de forma ordenada al objeto compartido.

```
run:
HiloA contador vale 400
HiloB contador vale 100
BUILD SUCCESSFUL (total time: 9 seconds)
```

Aún así se debe evitar la sincronización de bloques de código y sustituirlo siempre que sea posible por la sincronización de métodos, lo que implica que no es posible invocar dos métodos sincronizados del mismo objeto a la vez.

Para sincronizar un método simplemente añadimos la palabra clave synchronized a su declaración:

```
public syncrhonized void nombreMetodo() {
:    :    : :
}
```

Convertir un método en sincronizado tiene 2 efectos:

- Evita que dos invocaciones de métodos sincronizados del mismo objeto se mezclen. Cuando un hilo ejecuta un método sincronizado de un objeto, todos los hilos que invoquen métodos sincronizados del objeto se bloquearán hasta que el primer hilo termine con el objeto.
- Al terminar un método sincronizado, se garantiza que todos los hilos verán los cambios realizados sobre el objeto.

**Ejemplo:** si un hilo escribe en un fichero "El Quijote" y el otro escribe "Hamlet", al final quedarán todas las letras entremezcladas. Para conseguir que esto no pase, debemos sincronizar esa escritura para que escriba uno y cuando acabe el otro. Para ello se usa la palabra reservada **synchronized**:

```
      synchronized (fichero)
      {
            fichero.println("En    un    lugar    de    la    Mancha...");
      }
```

Al poner **synchronized(fichero)** marcamos **fichero** como ocupado desde que se abren las llaves de después hasta que se cierran. Cuando el segundo hilo intenta también su **synchronized(fichero)**, se queda ahí bloqueado, en espera que acabe el primero con **fichero**.

**synchronized** comprueba si fichero está o no ocupado. Si está ocupado, se queda esperando hasta que esté libre. Si está libre o una vez que esté libre, lo marca como ocupado y sigue el código.

**Ejemplo10:** Creamos un objeto llamado cuenta sobre el que se realizan acciones de sacar dinero. Al crear hilos sobre la cuenta puede darse la situación de sacar dinero cuando ya no hay efectivo en la cuenta. Debemos crear el método synchronized para bloquearlo y que no pueda ser llamado desde otro hilo hasta que no se libere.

```
package ejemplo10;

public class Ejemplo10 {

    public static void main(String[] args) {
        Cuenta c = new Cuenta(40);
        SacarDinero h1 = new SacarDinero("Ana", c);
        SacarDinero h2 = new SacarDinero("Juan", c);

        h1.start();
```

```
        h2.start();
    }
}

class Cuenta {

    private int saldo;

    Cuenta(int s) {
        saldo = s;
    }

    int getSaldo() {
        return saldo;
    }

    void restar(int cantidad) {
        saldo = saldo - cantidad;
    }

    void RetirarDinero(int cant, String nom) {
        if (getSaldo() >= cant) {
            System.out.println(nom + ": SE VA A RETIRAR SALDO (ACTUAL ES: " +
getSaldo() + ")");

            restar(cant);

            System.out.println("\t" + nom + " retira =>" + cant + " ACTUAL("
+ getSaldo() + ")");
        } else {
            System.out.println(nom  +  "  No  puede  retirar  dinero,  NO  HAY
SALDO(" + getSaldo() + ")");
        }
        if (getSaldo() < 0) {
            System.out.println("SALDO NEGATIVO => " + getSaldo());
        }
    }
}

class SacarDinero extends Thread {

    private Cuenta c;

    public SacarDinero(String n, Cuenta c) {
        super(n);
        this.c = c;
    }

    @Override
    public void run() {
        for (int x = 1; x <= 4; x++) {
            try {
```

```
                Thread.sleep((int) (Math.random() * 1000));
            } catch (InterruptedException ex) {
            }
            c.RetirarDinero(10, getName());
        }
    }
}
```

Salida: Se ha podido sacar efectivo aun sin haber dinero pues en el momento se consulto el saldo el hilo anterior no había retirado el efectivo que necesitaba.

```
run:
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
       Juan retira =>10 ACTUAL(20)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 20)
       Ana retira =>10 ACTUAL(20)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 20)
       Ana retira =>10 ACTUAL(10)
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 10)
       Juan retira =>10 ACTUAL(10)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 10)
       Juan retira =>10 ACTUAL(-10)
SALDO NEGATIVO => -10
       Ana retira =>10 ACTUAL(-10)
SALDO NEGATIVO => -10
Ana No puede retirar dinero, NO HAY SALDO(-10)
SALDO NEGATIVO => -10
Juan No puede retirar dinero, NO HAY SALDO(-10)
SALDO NEGATIVO => -10
BUILD SUCCESSFUL (total time: 1 second)
```

Para evitar el problema bloqueamos con **synchronized** el método RetirarDinero() de manera que al ser llamado por un hilo no pueda ser ejecutado por otro hasta que finalice.

```
synchronized void RetirarDinero(int cant, String nom) {
:      :      :      :      :      :
}
```

Salida:

```
run:
Ana: SE VA A RETIRAR SALDO (ACTUAL ES: 40)
       Ana retira =>10 ACTUAL(30)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 30)
       Juan retira =>10 ACTUAL(20)
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 20)
       Juan retira =>10 ACTUAL(10)
```

```
Juan: SE VA A RETIRAR SALDO (ACTUAL ES: 10)
      Juan retira =>10 ACTUAL(0)
Juan No puede retirar dinero, NO HAY SALDO(0)
Ana No puede retirar dinero, NO HAY SALDO(0)
Ana No puede retirar dinero, NO HAY SALDO(0)
Ana No puede retirar dinero, NO HAY SALDO(0)
BUILD SUCCESSFUL (total time: 2 seconds)
```

## 8.-Bloqueo de Hilos.

A veces nos interesa que un hilo se quede bloqueado a la espera de que ocurra algún evento, como la llegada de un dato para tratar o que el usuario termine de escribir algo en una interface de usuario.

Para poder coordinar la comunicación entre hilos utilizaremos los métodos wait(), notify() y notifyAll():

- **wait().** Permite suspender un hilo hasta que otro hilo llame al método notify() o notifyAll() del mismo objeto. Dicho hilo debe estar marcado como ocupado por medio de un **synchronized**. Si no se hace así, saltará una excepción.
- **notify().** Despierta de forma arbitraria a uno sólo de los hilos que realizó una llamada a wait().
- **notifyAll().** Despierta a todos los hilos que realizaron una llamada a wait().

Por ejemplo, si tenemos un hilo que permite retirar datos de una lista y si no hay datos, quiere esperar a que los haya, el hilo debe sincronizar el objeto haciendo algo como esto:

```
synchronized(lista) {
     if (lista.size()==0){
          lista.wait();
     }
     dato = lista.get(0);
     lista.remove(0);
}
```

Los pasos realizados son;

1.- **synchronized(lista)** para "apropiarnos" del objeto lista.
2.- Si no hay datos, hacemos el **lista.wait()**. Una vez que nos metemos en el **wait()**, el objeto lista queda marcado como "desocupado", de forma que otros hilos pueden usarlo.
3.- Cuando despertemos y salgamos del **wait()**, volverá a marcarse como "ocupado."
4.- Continuará con el proceso.

Para que nuestro hilo se desbloquee y salga del wait, se debe llamar al método **notify()** desde otro hilo que este en ejecución**.** Siguiendo con el ejemplo:

```
synchronized(lista){
   lista.add(dato);
   lista.notify();
}
```

Los métodos **wait()** y **notify()** funcionan como una lista de espera. Si varios hilos van llamando a **wait()** quedan bloqueados y en una lista de espera, de forma que el primero que llamó a **wait()** es el primero de la lista y el último es el último.

Cada llamada a **notify()** despierta solo al primer hilo en la lista de espera.

Si hacemos varios **notify()** antes de que haya hilos en espera, quedan marcados todos esos **notify()**, de forma que los siguientes hilos que hagan **wait()** no se quedaran bloqueados.

En resumen, **wait()** y **notify()** funcionan como un contador. Cada **wait()** mira el contador y si es cero o menos se queda bloqueado. Cuando se desbloquea decrementa el contador. Cada notify() incrementa el contador y si se hace 0 o positivo, despierta al primer hilo de la cola.

Si queremos despertar todos los hilos usaremos **notifyAll()**.

Un símil para entenderlo sería una mesa en la que hay gente que pone caramelos y gente que los recoge. La gente son los hilos. Los que van a coger caramelos (hacen **wait()**) se ponen en una cola delante de la mesa, cogen un caramelo y se van. Si no hay caramelos, esperan que los haya y forman una cola. Otras personas ponen un caramelo en la mesa (hacen **notify()**). El número de caramelos en la mesa es el contador que mencionábamos.

Un hilo puede salir del **wait()** sin necesidad de que nadie haga **notify()**. Esta situación se da cuando se produce algún tipo de interrupción. En el caso de java es fácil provocar una interrupción llamando al método **interrupt()** del hilo.

Por ejemplo, si el **hiloLector** está bloqueado en un **wait()** esperando un dato, podemos interrumpirle con: hiloLector.interrupt();

El **hiloLector** saldrá del **wait()** y se encontrará con que no hay datos en la lista. Sabrá que alguien le ha interrumpido y hará lo que tenga que hacer en ese caso.

## 9.-Modelo Productor - Consumidor.

Un problema típico de sincronización es el que representa el modelo Productor-Consumidor. Se produce cuando uno o más hilos producen datos a procesar y otros hilos los consumen. El problema surge cuando el productor produce datos más rápido que el consumidor los consuma. Igualmente el consumidor puede consumir más rápido que el productor produce.

Por ejemplo, imaginemos una aplicación donde un hilo (el productor) escribe datos en un fichero, mientras que un segundo hilo (el consumidor) lee los datos del mismo fichero. Los hilos comparten el mismo recurso (el fichero) y deben sincronizarse para realizar la tarea correctamente.

Una **Buena práctica** es "ocultar" el tema de la sincronización a los hilos, de forma que no dependamos de que el programador se acuerde de implementar su hilo correctamente (llamada a **synchronized** y llamada a **wait()** y **notify()**).

Podemos meter lista de datos dentro de una clase y poner dos métodos **synchronized** para añadir y recoger datos, con el **wait()** y el **notify()**dentro.

```
public class MiListaSincronizada {
    private ArrayList<Integer> lista;

    public MiListaSincronizada(){
        lista = new ArrayList<>();
    }

    public synchronized void addDato(int dato){
        lista.add(dato);
        lista.notify();
    }

    public synchronized int getDato() throws InterruptedException{
        if(lista.size() == 0){
            wait();
        }

        int dato = lista.get(0);
        lista.remove(0);
        return dato;
    }
}
```

Los hilos ya no deben preocuparse de nada. El hilo que espera por los datos hace esto quedándose bloqueado hasta que haya algún dato disponible.

```
int dato = listaSincronizada.getDato();
```

Mientras, el hilo que guarda datos sólo tiene que hacer esto otro:

```
listaSincronizada.addDato(dato);
```

**Ejemplo11:** Un ejemplo típico en que dos procesos necesitan sincronizarse es el caso en que un hilo produzca algún tipo de información que es procesada por otro hilo. La clase del objeto que se ejecutará en el primer hilo le denominaremos Productor y la clase del objeto que se ejecutará en el segundo hilo, Consumidor.

La clase Productor podría tener el siguiente aspecto:

```
public class Productor implements Runnable {
    private Contenedor contenedor;

    public Productor (Contenedor c) {
        contenedor = c;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            contenedor.put(i);
            try {
                sleep((int)(Math.random() * 100));
```

```
            } catch (InterruptedException e) { }
        }
    }
}
```

Productor tiene una variable contenedor, la cual es una referencia a un objeto Contenedor, que sirve para almacenar los datos que va produciendo. El método run genera aleatoriamente el dato y lo coloca en el contenedor con el método put. Después espera una cantidad de tiempo aleatoria (hasta 100 milisegundos) con el método sleep. Productor no se preocupa de si el dato ya ha sido consumido o no. Simplemente lo coloca en el contenedor.

El Consumidor, por su parte podría tener el siguiente aspecto:

```
public class Consumidor implements Runnable {
     private Contenedor contenedor;
     public Consumidor (Contenedor c) {
        contenedor= c;
     }
     public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
          value = contenedor.get();
        }
     }
}
```

El constructor es equivalente al del Productor. El método run, simplemente recupera el dato del contenedor con el método get y lo muestra en la consola. Tampoco el consumidor se preocupa de si el dato está ya disponible en el contenedor o no.

Productor y Consumidor se usarían desde un método main de la siguiente forma:

```
public class Principal {
    public static void main(String[] args) {
        Contenedor c = new Contenedor ();
        Productor productor = new Productor (c);
        Consumidor consumidor = new Consumidor (c);
     Thread hiloProductor = new Thread(productor);
     Thread hiloConsumidor = new Thread(consumidor);

        hiloProductor.start();
        hiloConsumidor.start();
    }
}
```

Simplemente se crean los objetos, contenedor, productor, consumidor y los dos hilos. Después los iniciamos con el método start.

La sincronización que permite a productor y consumidor operar correctamente, es decir que hace que consumidor espere hasta que haya un dato disponible, y que productor no genere

uno nuevo hasta que haya sido consumido está en la clase Contenedor, que tiene el siguiente aspecto:

```
public class Contenedor {
     private int dato;
     private boolean hayDato = false;

     public synchronized int get() {
          while (hayDato == false) {
             try {
             // espera a que el productor coloque un valor
                    wait();
             }
             catch (InterruptedException e) { }
          }
          hayDato = false;
          System.out.println("Consumidor. get: " + dato);

          // notificar que el valor ha sido consumido
          notifyAll();
          return dato;
     }

     public synchronized void put(int valor) {
          while (hayDato == true) {
             try {
                    // espera a que se consuma el dato
                    wait();
             } catch (InterruptedException e) { }
          }
          dato = valor;
          hayDato = true;
          System.out.println("Productor. put: " + dato);

          // notificar que ya hay dato.
          notifyAll();
     }
}
```

El atributo dato es el que contiene el valor que se almacena con put y se devuelve con get. El atributo hayDato es un flag interno que indica si el objeto contiene dato o no.

En el método put, antes de almacenar el valor en dato hay que asegurarse de que el valor anterior ha sido consumido. Si todavía hay valor ( hayDato es true) se suspende la ejecución del hilo mediante el método wait.

Invocando wait (que es un método de la clase Object) se suspende el hilo indefinidamente hasta que alguien le envíe una 'señal' con el método notify o notifyAll. Cuando esto se produce (veremos que el notify lo produce el método get) el método continua, asume que el dato ya se

ha consumido, almacena el valor en dato y envía a su vez un notifyAll para notificar a su vez que hay un dato disponible.

Por su parte, el método get comprueba si hay dato disponible (no lo hay si hayDato es false) y si no lo hay espera hasta que le avisen (método wait). Una vez ha sido notificado (desde el método put) cambia el flag y devuelve el dato, pero antes notifica a put de que el dato ya ha sido consumido, y por tanto se puede almacenar otro. Con esto podemos ver que la sincronización se lleva a cabo usando los métodos wait y notifyAll.

Existe además otro componente básico en el ejemplo. Los objetos productor y consumidor utilizan un recurso compartido que es el objeto contenedor. Si mientras el productor llama al método put y este se encuentra cambiando las variables miembro dato y hayDato, el consumidor llamara al método get y este a su vez empezara a cambiar estos valores podrían producirse resultados inesperados.

Interesa, por tanto que mientras se esté ejecutando el método put nadie más acceda a las variables miembro del objeto. Esto se consigue con la palabra synchronized en la declaración del método. Cuando la máquina virtual inicia la ejecución de un método con este modificador adquiere un bloqueo en el objeto sobre el que se ejecuta el método que impide que nadie más inicie la ejecución en ese objeto de otro método que también esté declarado como synchronized. En nuestro ejemplo cuando comienza el método put se bloquea el objeto de tal forma que si alguien intenta invocar el método get o put (ambos son synchronized) quedará en espera hasta que el bloqueo se libere (cuando termine la ejecución del método). Este mecanismo garantiza que los objetos compartidos mantienen la consistencia.

Este método de gestionar los bloqueos implica que:

- Es responsabilidad del programador pensar y gestionar los bloqueos (A veces es una pesada responsabilidad).
- Los métodos synchronized son más costosos en el sentido de que adquirir y liberar los bloqueos consume tiempo (este es el motivo por el que no están sincronizados por defecto todos los métodos).
- Conviene evitar en lo posible el uso de objetos compartidos. Resultan difíciles de manejar.

**Código Completo:**

```
import static java.lang.Thread.*;
public class Ejemplo12 {
    public static void main(String[] args) {
        Contenedor c = new Contenedor ();
        Productor productor = new Productor (c);
        Consumidor consumidor = new Consumidor (c);
        Thread hiloProductor = new Thread(productor);
        Thread hiloConsumidor = new Thread(consumidor);
        hiloProductor.start();
        hiloConsumidor.start();
    }
```

```
}

class Productor implements Runnable {
    private Contenedor contenedor;

    public Productor (Contenedor c) {
        contenedor = c;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            contenedor.put(i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}

class Consumidor implements Runnable {
     private Contenedor contenedor;
     public Consumidor (Contenedor c) {
        contenedor= c;
     }
        @Override
     public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
          value = contenedor.get();
        }
     }
}

class Contenedor {
     private int dato;
     private boolean hayDato = false;

     public synchronized int get() {
          while (hayDato == false) {
             try {
             // espera a que el productor coloque un valor
                   wait();
             }
             catch (InterruptedException e) { }
          }
          hayDato = false;
          System.out.println("Consumidor. get: " + dato);

          // notificar que el valor ha sido consumido
          notifyAll();
          return dato;
```

```
    }

    public synchronized void put(int valor) {
        while (hayDato == true) {
          try {
                // espera a que se consuma el dato
                wait();
          } catch (InterruptedException e) { }
        }
        dato = valor;
        hayDato = true;
        System.out.println("Productor. put: " + dato);

        // notificar que ya hay dato.
        notifyAll();
    }
}
```

**Salida:**

```
run:
Productor. put: 0
Consumidor. get: 0
Productor. put: 1
Consumidor. get: 1
Productor. put: 2
Consumidor. get: 2
Productor. put: 3
Consumidor. get: 3
Productor. put: 4
Consumidor. get: 4
Productor. put: 5
Consumidor. get: 5
Productor. put: 6
Consumidor. get: 6
Productor. put: 7
Consumidor. get: 7
Productor. put: 8
Consumidor. get: 8
Productor. put: 9
Consumidor. get: 9
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Ejemplo12**: Tenemos dos objetos: Cola de Parking y Coche. La Cola de parking tiene métodos sincronizados que permiten entrar y salir coches del parking. Solo puede entrar un vehículo al parking si hay plazas libres. Si no hay plazas libres el vehículo deberá esperar a que haya. Por otro lado sólo pueden salir coche del parking si hay aparcado algún vehículo. Se crearán dos hilos, uno para aparcar y otro para salir coches.

```
import java.util.ArrayList;
public class Ejemplo11 {
    public static void main(String[] args) {
        Cola miCola = new Cola();
        HiloA miHiloA = new HiloA(miCola);
        miHiloA.start();
        HiloB miHiloB = new HiloB(miCola);
        miHiloB.start();
    }
}
class HiloA extends Thread {
    Cola miCola;
    static int plazas = 10;
```

```
    public HiloA(Cola miCola) {
        this.miCola = miCola;
    }
    @Override
    public void run() {
        for (int i = 1; i <= 40; i++) {
            Coche miCoche = new Coche(i);
            try {
                miCola.entraCoche(miCoche);
            } catch(InterruptedException e){}
            try {
                Thread.sleep((int)(Math.random()*1000));
            } catch (InterruptedException e) {}
        }
    }
}
class HiloB extends Thread {
    Cola miCola;
    static int plazas = 10;
    public HiloB(Cola miCola) {
        this.miCola = miCola;
    }
    @Override
    public void run() {
        for (int i = 1; i <= 40; i++) {
            try {
                Coche miCoche = miCola.saleCoche();
            } catch (InterruptedException ex) { }
            try {
                Thread.sleep((int)(Math.random()*1500));
            } catch (InterruptedException e) {}
        }
    }
}
class Cola {
    private ArrayList<Coche> parking;
    static int plazas = 10;
    public Cola() {
        parking = new ArrayList<>();
    }
    public    synchronized    void    entraCoche(Coche    miCoche)    throws
InterruptedException {
        int libres = plazas-parking.size();
        if (libres<=0){
            wait();
        }
        parking.add(miCoche);
        System.out.println("Vehiculo Aparcado. Plazas: "+
             (plazas-parking.size()));
        notify();
    }
    public synchronized Coche saleCoche() throws InterruptedException {
```

```
        if (parking.size() == 0) {
            wait();
        }
        Coche miCoche = parking.get(0);
        parking.remove(0);
        System.out.println("Vehiculo sale del Parking. Plazas: "
            +(plazas-parking.size()));
        notify();
        return miCoche;
    }
}
class Coche {
    private int numCoche;
    public Coche(int numCoche) {
        this.numCoche = numCoche;
    }
    public int getMatricula() {
        return numCoche;
    }
    public void setMatricula(int numCoche) {
        this.numCoche = numCoche;
    }
}
```

Salida:

```
run:
Vehiculo Aparcado. Plazas: 9
Vehiculo sale del Parking. Plazas: 10
Vehiculo Aparcado. Plazas: 9
Vehiculo sale del Parking. Plazas: 10
Vehiculo Aparcado. Plazas: 9
Vehiculo sale del Parking. Plazas: 10
Vehiculo Aparcado. Plazas: 9
Vehiculo sale del Parking. Plazas: 10
Vehiculo Aparcado. Plazas: 9
Vehiculo Aparcado. Plazas: 8
Vehiculo Aparcado. Plazas: 7
Vehiculo Aparcado. Plazas: 6
Vehiculo Aparcado. Plazas: 5
Vehiculo Aparcado. Plazas: 4
Vehiculo sale del Parking. Plazas: 5
Vehiculo Aparcado. Plazas: 4
Vehiculo sale del Parking. Plazas: 5
Vehiculo Aparcado. Plazas: 4
Vehiculo Aparcado. Plazas: 3
Vehiculo sale del Parking. Plazas: 4
Vehiculo sale del Parking. Plazas: 5
Vehiculo Aparcado. Plazas: 4
Vehiculo sale del Parking. Plazas: 5
Vehiculo Aparcado. Plazas: 4
Vehiculo sale del Parking. Plazas: 5
Vehiculo sale del Parking. Plazas: 6
Vehiculo sale del Parking. Plazas: 7
Vehiculo Aparcado. Plazas: 6
Vehiculo sale del Parking. Plazas: 7
Vehiculo Aparcado. Plazas: 6
Vehiculo Aparcado. Plazas: 5
Vehiculo Aparcado. Plazas: 4
Vehiculo sale del Parking. Plazas: 5
Vehiculo Aparcado. Plazas: 4
Vehiculo Aparcado. Plazas: 3
Vehiculo Aparcado. Plazas: 2
Vehiculo Aparcado. Plazas: 1
Vehiculo sale del Parking. Plazas: 2
Vehiculo Aparcado. Plazas: 1
Vehiculo Aparcado. Plazas: 0
Vehiculo sale del Parking. Plazas: 1
Vehiculo Aparcado. Plazas: 0
Vehiculo sale del Parking. Plazas: 1
Vehiculo Aparcado. Plazas: 0
Vehiculo sale del Parking. Plazas: 1
Vehiculo Aparcado. Plazas: 0
Vehiculo sale del Parking. Plazas: 1
Vehiculo Aparcado. Plazas: 0
Vehiculo sale del Parking. Plazas: 1
Vehiculo Aparcado. Plazas: 0
Vehiculo sale del Parking. Plazas: 1
Vehiculo sale del Parking. Plazas: 2
Vehiculo Aparcado. Plazas: 1
Vehiculo Aparcado. Plazas: 0
Vehiculo sale del Parking. Plazas: 1
Vehiculo Aparcado. Plazas: 0
Vehiculo sale del Parking. Plazas: 1
Vehiculo Aparcado. Plazas: 0
Vehiculo sale del Parking. Plazas: 1
Vehiculo sale del Parking. Plazas: 2
Vehiculo Aparcado. Plazas: 1
Vehiculo sale del Parking. Plazas: 2
Vehiculo sale del Parking. Plazas: 3
Vehiculo Aparcado. Plazas: 2
Vehiculo Aparcado. Plazas: 1
```

```
Vehiculo sale del Parking. Plazas: 2
Vehiculo sale del Parking. Plazas: 3
Vehiculo Aparcado. Plazas: 2
Vehiculo Aparcado. Plazas: 1
Vehiculo sale del Parking. Plazas: 2
Vehiculo Aparcado. Plazas: 1
Vehiculo sale del Parking. Plazas: 2
Vehiculo sale del Parking. Plazas: 3
```

```
Vehiculo sale del Parking. Plazas: 4
Vehiculo sale del Parking. Plazas: 5
Vehiculo sale del Parking. Plazas: 6
Vehiculo sale del Parking. Plazas: 7
Vehiculo sale del Parking. Plazas: 8
Vehiculo sale del Parking. Plazas: 9
Vehiculo sale del Parking. Plazas: 10
BUILD SUCCESSFUL (total time: 26 seconds)
```

**Ejemplo12 con Comentarios:**

```
package productorconsumidor;

import java.util.ArrayList;

/*
Tenemos dos objetos: Cola de Parking y Coche. La Cola de parking tiene métodos
sincronizados que permiten entrar y salir coches del parking. Solo puede entrar un vehículo al
parking si hay plazas libres. Si no hay plazas libres el vehículo deberá esperar a que haya. Por
otro lado sólo pueden salir coche del parking si hay aparcado algún vehículo. Se crearán dos
hilos, uno para aparcar y otro para salir coches.
*/
public class ProductorConsumidor {
    public static void main(String[] args) {
        //Creamos instancia de Cola (Objeto Compartido)
        // Creamos los Hilos de entrar y salir coche que se repiten 40 veces
        // Ejecutamos los hilos
        Cola miCola = new Cola();
        HiloSale miHiloSale = new HiloSale(miCola);
        HiloEntra miHiloEntra = new HiloEntra(miCola);
        miHiloSale.start();
        miHiloEntra.start();
    }
}

// Creamos los hilos que ejecuta los métodos que permitem entrar y salir coches del parking

class HiloSale extends Thread{
    // Instancia de Cola
    // Objeto compartido por ambos hilos
    Cola miCola;
    static int plazas = 10;
    // Constructor de Cola
    public HiloSale(Cola miCola){
        this.miCola=miCola;
    }
```

```
    @Override
    // Método que ejecuta el hilo
    public void run(){
        // Bucle que ejecuta el hilo que permite salir coches 40 veces
        for(int i = 1; i <= 40; i++){
            // Lanzamos el método saleCoche()
            try{
                Coche miCoche = miCola.saleCoche();
            }catch(InterruptedException ex){}
            // Esperamos un tiempo aleatorio para salir coche del parking y
no sea inmediato
            try{
                Thread.sleep((int)(Math.random()*4000));
            }catch(InterruptedException ex){}
        }
    }
}

class HiloEntra extends Thread{
    // Instancia de Cola
    // Objeto compartido por ambos hilos
    Cola miCola;
    static int plazas =10;
    // Constructor de Cola
    public HiloEntra(Cola miCola){
        this.miCola = miCola;
    }
    @Override
    // Método que ejecuta el hilo
    public void run(){
        // Bucle que ejecuta el hilo que permite entrar coches 40 veces.
        for(int i = 1; i<=40; i++){
            // Creamos coche con num igual al valor del bucle 1,2,3....
            Coche miCoche = new Coche(i);
            // Lanzamos el método entraCoche
            try {
                miCola.entraCoche(miCoche);
            }catch(InterruptedException e){}
            // Esperamos un tiempo aleatorio que tarda entrar coche para que
no sea inmediato
            try {
                Thread.sleep((int)(Math.random()*3000));
            } catch(InterruptedException e){}
        }
    }
}

// Clase Cola Objeto compartido por los dos hilos.
class Cola{
    private ArrayList<Coche> parking;
    static int plazas = 10;
    public Cola(){
```

```
        parking = new ArrayList<>();
    }

    public synchronized void entraCoche(Coche miCoche) throws
InterruptedException {
        // Calculamos plazas libres del parking
        int libres = plazas - parking.size();
        // Si no hay plazas libres no puede entrar coche y el hilo debe
esperar
        if(libres<=0){
            wait();
        }
        // En cuanto hay plazas libres se añade el coche al parking
        parking.add(miCoche);
        // En este punto se puede haber ejecutado saleCoche por lo que el
valor de libres
        // podía no ser real. Mejor utilizar (plazas - parking.size())
        System.out.println("Vehiculo aparcado. Plazas: "+(plazas -
parking.size()));
        // Avisa al hilo saleCoche que puede salir un Coche
        notify();
    }

    public synchronized Coche saleCoche() throws InterruptedException {
        // Si no hay coches en el parking no pueden salir y hay que esperar
        if(parking.size()==0){
            wait();
        }
        // En cuanto hay un coche en el parking ya es posible que salga
        Coche miCoche = parking.get(0);
        parking.remove(0);
        System.out.println("Vehiculo sale del parking. Plazas: "+(plazas-
parking.size()));
        // Avisa que puede entrar un coche si fuera necesario
        notify();
        return miCoche;
    }
}

// Clase Coche
class Coche {
    private int numCoche;

    public Coche(int numCoche) {
        this.numCoche = numCoche;
    }

    public int getNumCoche() {
        return numCoche;
    }

    public void setNumCoche(int numCoche) {
```

```
        this.numCoche = numCoche;
    }

    @Override
    public String toString() {
        return "Coche{" + "numCoche=" + numCoche + '}';
    }


}
```