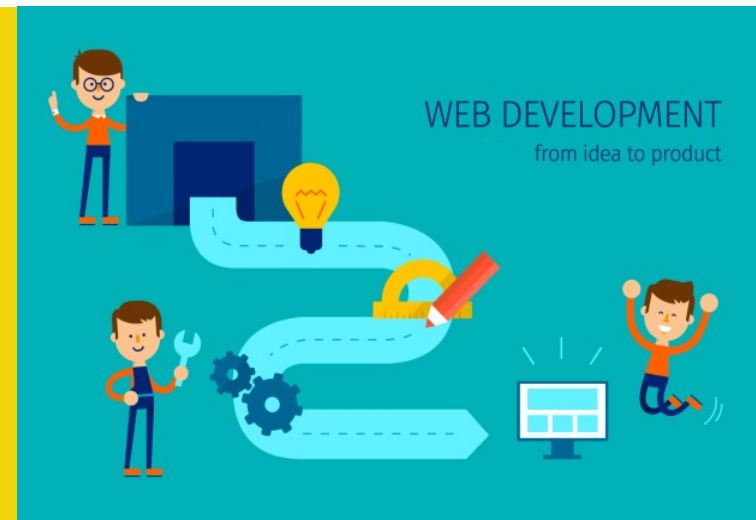


# Módulo 9

Código: 490

## Programación de Servicios y Procesos

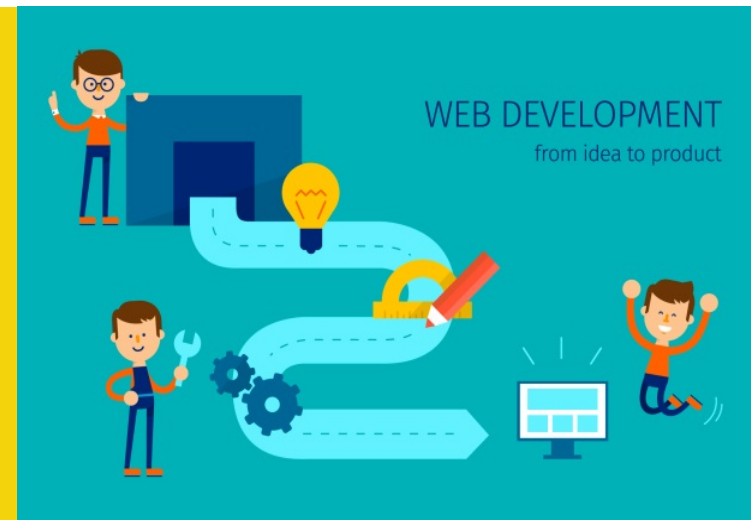
Técnico Superior en Desarrollo de  
Aplicaciones Multiplataforma



# UNIDAD 2

## Programación multihilo

Contenidos teóricos

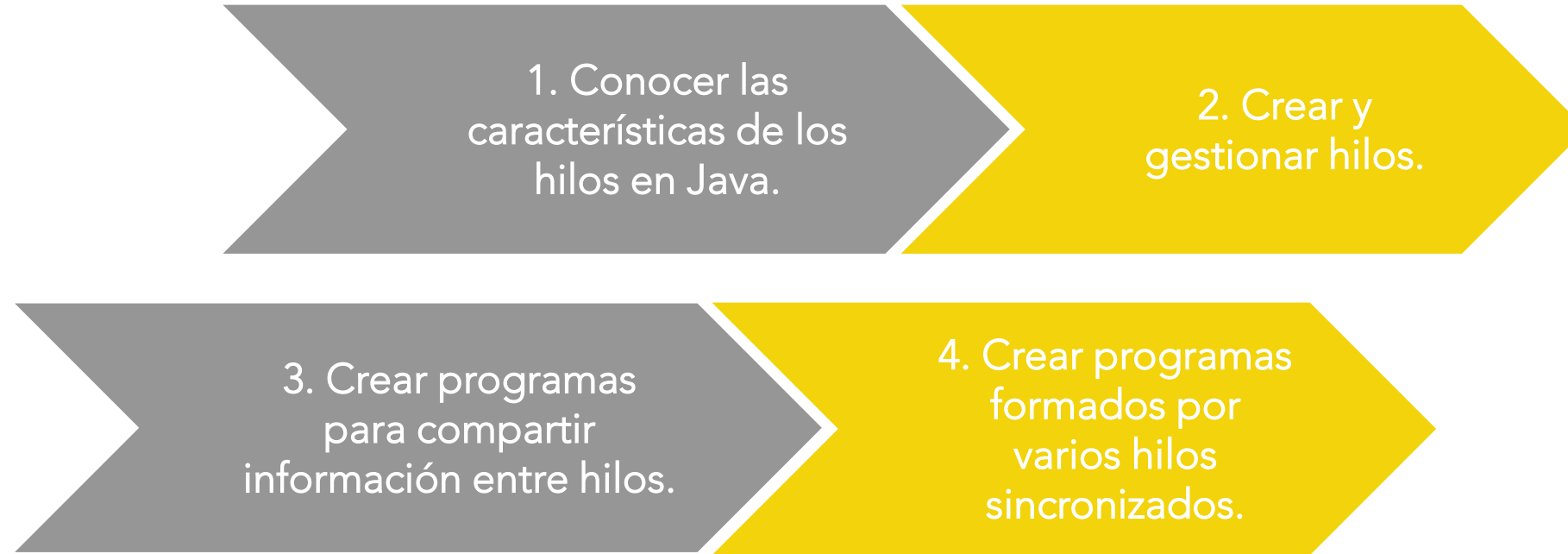


1. Objetivos generales de la unidad
2. Competencias y contribución en la unidad
3. Contenidos conceptuales y procedimentales
4. Evaluación
5. Distribución de los contenidos
6. Sesiones


# 1. Objetivos generales de la unidad

## Objetivos curriculares

---



## 2. Competencias y contribución en la unidad



n) Desarrollar aplicaciones multiproceso y multihilo empleando librerías y técnicas de programación específicas.

	<p>A través del estudio de los hilos. Aprendiendo a crear y gestionar hilos en programas Java.</p>	
--	--	--

### 3. Contenidos

CONTENIDOS CONCEPTUALES		CONTENIDOS PROCEDIMENTALES
Hilos	<ul style="list-style-type: none"><li>Definición.</li><li>Estados.</li><li>Librerías y clases para la programación de hilos en Java.</li><li>Gestión de hilos.</li></ul>	<ul style="list-style-type: none"><li>Qué son los hilos.</li><li>Ver los diferentes estados por los que pasa un hilo durante su vida.</li><li>Trabajar con el API de Java relacionada con la gestión de hilos.</li></ul>
Programación multihilo	<ul style="list-style-type: none"><li>Sincronización de hilos.</li><li>Comunicación entre hilos.</li><li>Gestión de prioridades</li></ul>	<ul style="list-style-type: none"><li>Utilización de la programación de varios hilos en un entorno multiproceso.</li><li>Ver la necesidad de sincronizar los diferentes hilos de un programa concurrente, y la forma de llevarlo a cabo.</li><li>En caso de que los diferentes hilos tengan prioridades de ejecución, forma de gestionarlo.</li><li>Ver la posibilidad de que diferentes hilos de ejecución compartan objetos.</li></ul>

## 4. Evaluación

Se han identificado situaciones en las que resulte útil la utilización de varios hilos en un programa.

Se han reconocido los mecanismos para crear, iniciar y finalizar hilos.

Se han programado aplicaciones que implementen varios hilos.

Se han identificado los posibles estados de ejecución de un hilo y programado aplicaciones que los gestionen.

Se han utilizado mecanismos para compartir información entre varios hilos de un mismo proceso.

Se han desarrollado programas formados por varios hilos sincronizados mediante técnicas específicas.

## 4. Evaluación

---

Se ha establecido y controlado la prioridad de cada uno de los hilos de ejecución.

Se han depurado y documentado los programas desarrollados.



## 1. PROGRAMACIÓN MULTITHILO

- Un **programa multihilo** contiene varias partes que se pueden ejecutar simultáneamente. Cada una de estas partes se llama **hilo** (*thread*), de tal forma que cada hilo tiene su ruta de ejecución independiente.
- Podemos decir entonces que la programación multihilo es una forma especializada de programación multitarea.



## 1. PROGRAMACIÓN MULTITHILO

- Diferencias entre la **multitarea basada en procesos** y la **basada en hilos**:
  - ✓ En la **multitarea basada en procesos**: es posible ejecutar dos o más programas al mismo tiempo. Por ejemplo, ejecutar Word al mismo tiempo que se navega por Internet. El proceso (programa) es la unidad más pequeña que gestiona el planificador de la CPU.
  - ✓ En la **multitarea basado en hilos**: un mismo programa puede realizar dos o más tareas a la vez, por ejemplo, Word puede estar pasando el corrector ortográfico a la vez que está imprimiendo. Estas 2 tareas se realizan con 2 hilos separados. Así pues el hilo es la unidad más pequeña de código ejecutable.
- Aunque los programas Java utilizan entornos multitarea basados en procesos, la multitarea basada en procesos no está bajo el control de Java. La multitarea multihilo si.

## 2. RECURSOS COMPARTIDOS POR LOS HILOS

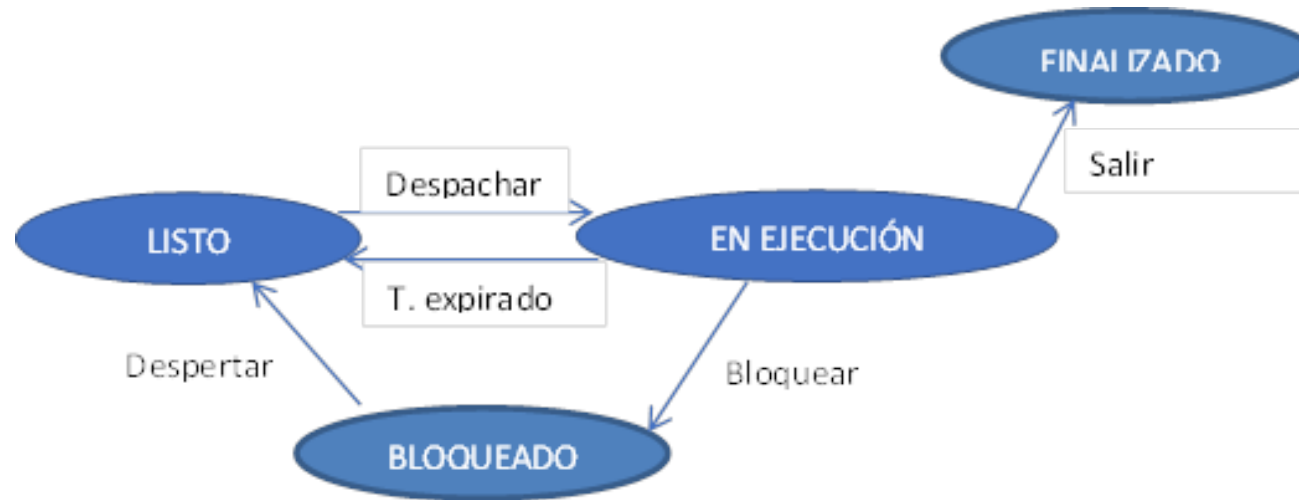
- Cada thread representa un proceso individual ejecutándose en el sistema. A veces se les llama procesos ligeros.
- Todos los hilos comparten los mismos recursos, al contrario que los procesos, en donde cada uno tiene su propia copia de código y datos (separados unos de otros):

PROCESO



- Cuando en un programa tenemos varios hilos ejecutándose simultáneamente, puede ocurrir que estos intenten acceder a los mismos datos, por ejemplo un fichero, pudiendo entorpecerse entre ellos.
- Para evitar estos problemas es necesario **sincronizar** los hilos.

## 3. ESTADOS DE UN HILO



- Estos son los estados principales de un hilo son:
  - ✓ **Listo**: el hilo está creado pero todavía no ha comenzado su ejecución.
  - ✓ **En ejecución**: cuando el hilo está ejecutándose (pasa a la CPU).
  - ✓ **Bloqueado**: cuando el hilo espera que finalice alguna tarea como por ejemplo una operación de E/S.
  - ✓ **Finalizado**: el hilo termina su ejecución, y ya no puede reanudarse.

## 3. ESTADOS DE UN HILO

- Es el S.O. el que controla los estados de un hilo, es decir, nosotros como programadores no tenemos que preocuparnos de cuando pasar un hilo a ejecución, o cuando bloquearlo. De estas tareas, obviamente, se encargará el sistema.
- Sin embargo, lo que sí es tarea nuestra es controlar la interacción entre los hilos usando los métodos adecuados, para evitar problemas como los siguientes:
  - ✓ **Interbloqueo.** Se produce cuando las peticiones y las esperas se entrelazan de forma que ningún hilo puede avanzar.
  - ✓ **Inanición.** Ningún hilo consigue hacer nada, por lo que hay que esperar a que el administrador del sistema detecte el interbloqueo y mate procesos (o hasta que se reinicie el equipo).

## 4. LIBRERÍAS Y CLASES PARA LA PROGRAMACIÓN DE HILOS EN JAVA

- Para crear un hilo en Java se pueden hacer dos cosas:
  1. Heredar de la clase **Thread**.
  2. Implementar la interfaz **Runnable**.
- Lo más aconsejable es utilizar **Runnable**, por diversos motivos:
  - ✓ Se consigue un código más limpio, separando el código en sí de la implementación de subprocesos.
  - ✓ No modificamos el comportamiento del hilo sino que le damos algo para que se ejecute.
  - ✓ Se consigue una mayor flexibilidad, ya que si heredamos de **Thread** la acción siempre estará en un **Thread**, mientras que si implementamos **Runnable** no tiene por qué, podemos ejecutar la tarea en un hilo, o pasarlo a algún tipo de servicio ejecutor, o simplemente pasarlo como una tarea dentro de una aplicación monotarea.

## 4. LIBRERÍAS Y CLASES PARA LA PROGRAMACIÓN DE HILOS EN JAVA

### Heredar de Thread

```
public class EjemploThread extends Thread {  
  
    public EjemploThread() {  
        super("EjemploThread");  
    }  
  
    @Override  
    public void run() {  
        //código  
    }  
  
}
```

- Crea un hilo con la instrucción:  
*new EjemploThread().start();*

### Implementar Runnable

```
public class EjemploRunnable implements Runnable {  
  
    @Override  
    public void run() {  
        //código  
    }  
  
}
```



- Crea un hilo con la instrucción:  
*new Thread(new EjemploRunnable()).start();*

## 4. LA CLASE THREAD

- Como vemos en los ejemplos anteriores, ambos enfoques crean un hilo instanciando un objeto de tipo **Thread**.
- La diferencia está, en como crear la clase que está habilitada para ser un hilo de ejecución.
- Es decir, la clase que herede de **Thread** o implemente **Runnable**, es la que encapsula el objeto que se puede ejecutar.
- La clase **Thread** define varios métodos que ayudan a administrar los hilos:



## 4. LA CLASE THREAD

Método	Descripción
String getName()	Obtiene el nombre de un hilo.
int getPriority	Obtiene la prioridad de un hilo.
boolean isAlive()	Determina si un hilo todavía se está ejecutando.
void join()	Espera a que termine un hilo.
run()	Punto de entrada para el hilo.
void sleep(long milisegundos)	Suspende un hilo durante un período específico de milisegundos.
start()	Inicia un hilo llamando a su método run().

## 5. IMPLEMENTACIÓN DE HILOS CON RUNNABLE

- La interfaz **Runnable** únicamente tiene el método `run()`.
- Este método establece el punto de entrada para otro hilo de ejecución concurrente en nuestro programa.
- Dentro de él colocaremos el código que constituye el nuevo hilo.
- Este hilo terminará cuando retorne `run()`.
- Una vez que hemos creado la clase que será un nuevo hilo de ejecución, para crear ese hilo, instanciamos un objeto de tipo **Thread**:

*Thread hilo = new Thread(objetoRunnable);*

- Donde *objetoRunnable* es una instancia de la clase que implementa la interfaz **Runnable**.
- Una vez creado el nuevo hilo no comenzará a ejecutarse hasta que llame a su método `start()`, que lo que hace es llamar al método `run()`:

*hilo.start();*

- Es habitual almacenar todos los hilos de ejecución de un programa en un vector.

## 5. IMPLEMENTACIÓN DE HILOS CON RUNNABLE

- Ejemplo:

```
public class MiHilo implements Runnable {  
  
    private String nombreHilo;  
  
    public MiHilo(String nombre) {  
        this.nombreHilo = nombre;  
    }  
  
    //Punto de entrada del hilo  
    //Los hilos comienzan a ejecutarse aquí  
    @Override  
    public void run() {  
        System.out.println("Comenzando ejecución del hilo: " + nombreHilo);  
        try {  
            for (int cont = 0; cont < 10; cont++) {  
                Thread.sleep(400);  
                System.out.println("Hilo: " + nombreHilo + " Contando: " + cont);  
            }  
        } catch (InterruptedException ex) {  
            System.out.println(nombreHilo + " Interrumpido.");  
        }  
        System.out.println("Terminando hilo: " + nombreHilo);  
    }  
}
```

## 5. IMPLEMENTACIÓN DE HILOS CON RUNNABLE

```
public class LanzadorHilos {  
  
    public static void main(String[] args) throws InterruptedException {  
  
        System.out.println("Hilo principal iniciando.");  
        //inicialmente construimos un objeto MiHilo.  
        MiHilo obj = new MiHilo("#1");  
        //después construimos un hilo de ese objeto.  
        Thread hilol = new Thread(obj);  
        //y finalmente iniciamos la ejecución de ese hilo  
        hilol.start();  
        for (int i = 0; i < 50; i++) {  
            System.out.print(" .");  
            Thread.sleep(100);  
        }  
  
        System.out.println("Hilo principal finalizado.");  
    }  
}
```

## 5. IMPLEMENTACIÓN DE HILOS CON RUNNABLE

- Analicemos el código:
  - ✓ Todo programa/proceso tiene al menos un hilo de ejecución, llamado hilo principal. Desde este hilo principal creamos otros hilos. En nuestro caso el hilo principal es el "hilo main".
  - ✓ En el hilo principal primero se crea un objeto **MiHilo**, que luego se usa para construir un objeto **Thread**. Esto es posible porque MiHilo implementa Runnable.
  - ✓ La ejecución del nuevo hilo con *start()*, crea un hilo hijo y llama al método *run()* de éste. El método *start()* pone en estado "En ejecución" al hilo.
  - ✓ Después la ejecución vuelve al *main()* entrando en el bucle. Ambos hilos (main e hijo) continúan ejecutándose, compartiendo la CPU en sistemas de una sola CPU, hasta que terminan sus bucles. El resultado obtenido puede ser diferente dependiendo de la máquina en la que se ejecute.

## 5. IMPLEMENTACIÓN DE HILOS CON RUNNABLE

- El hilo main deberá esperar a que termine el hilo hijo. Esto se puede hacer:
  - ✓ Utilizando diferencias de tiempos entre los 2 hilos, con llamadas al método *sleep()*, para asegurar que el hilo main() finaliza el último. Ésta última forma no es la habitual en la práctica.
  - ✓ Con el método *join()*. Con este método hacemos que el hilo main espere a que finalice el hilo hijo.
- En este primer ejemplo hemos creado un único hilo hijo, pero podemos crear tantos hilos hijo como necesitemos.

## 5. IMPLEMENTACIÓN DE HILOS CON RUNNABLE

- Ejemplo de creación de 5 hilos hijo:

```
public class LanzadorHilos {  
    public static void main(String[] args) throws InterruptedException {  
        final int NUM_HILOS = 5;  
        System.out.println("Hilo principal iniciando.");  
  
        MiHilo obj;  
        for (int i = 0; i < NUM_HILOS; i++) {  
            obj = new MiHilo("#" + i);  
            Thread hiloHijo = new Thread(obj);  
            hiloHijo.start();  
        }  
        for (int i = 0; i < 50; i++) {  
            System.out.print(" .");  
            Thread.sleep(100);  
        }  
        System.out.println();  
        System.out.println("Hilo principal finalizado.");  
    }  
}
```

## 6. GESTIÓN DE HILOS

- Como hemos visto, las operaciones básicas a realizar con un hilo son:
  - ✓ Creación: clase **Thread** / interfaz **Runnable**
  - ✓ Inicio: método *start()* ► método *run()*
  - ✓ Finalización: fin método *run()*
- Pero también podemos realizar otras operaciones con los hilos que nos pueden ser de utilidad:
  - ✓ Asignar un nombre al hilo: método *setName("nombre")*.
  - ✓ Obtener el objeto asociado al hilo de ejecución: método *currentThread()*.
  - ✓ Asignar prioridad al hilo: método *setPriority(prioridad)*.  
Donde prioridad puede estar dentro del rango [MIN\_PRIORITY, MAX\_PRIORITY].
  - ✓ Finalizar un hilo de ejecución llamando al método *join*. Este método devuelve el control al hilo principal que lanzó el hilo secundario.



## 6. GESTIÓN DE HILOS

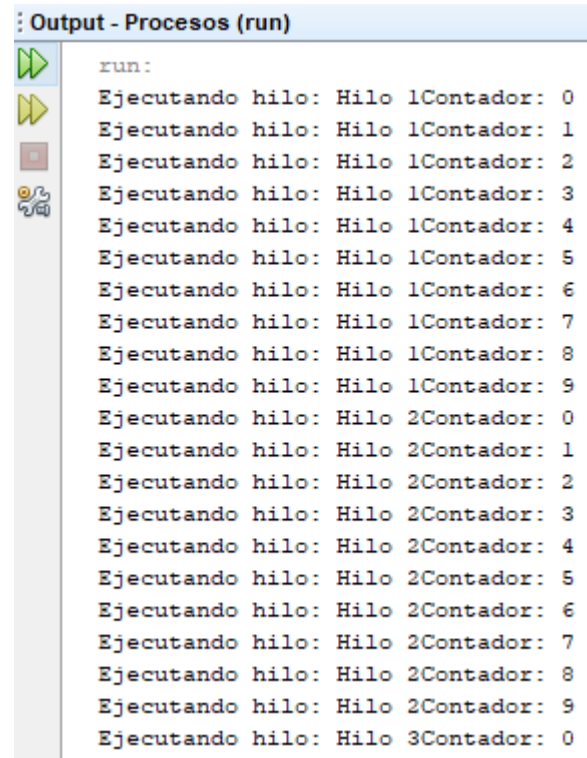
- Veamos un ejemplo:

```
public class NuevoHilo implements Runnable {  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println("Ejecutando hilo: " + Thread.currentThread().getName() + "Contador: " + i)  
        }  
    }  
}
```

```
public class LanzadorHilos {  
  
    public static void main(String[] args) {  
        NuevoHilo arrayHilos[] = new NuevoHilo[5];  
        for (int i = 0; i < 5; i++) {  
            arrayHilos[i] = new NuevoHilo();  
            Thread hilo = new Thread(arrayHilos[i]);  
            hilo.setName("Hilo " + (i+1));  
            if (i == 0) {  
                hilo.setPriority(Thread.MAX_PRIORITY);  
            }  
            hilo.start();  
        }  
    }  
}
```

## 6. GESTIÓN DE HILOS

- Si ejecutamos el programa varias veces, el orden de ejecución de los hilos variará.

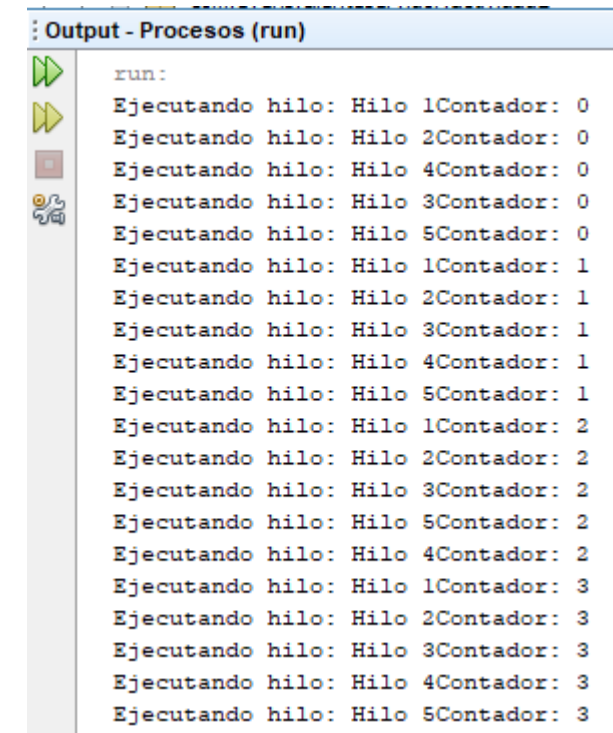


```
run:
Ejecutando hilo: Hilo 1Contador: 0
Ejecutando hilo: Hilo 1Contador: 1
Ejecutando hilo: Hilo 1Contador: 2
Ejecutando hilo: Hilo 1Contador: 3
Ejecutando hilo: Hilo 1Contador: 4
Ejecutando hilo: Hilo 1Contador: 5
Ejecutando hilo: Hilo 1Contador: 6
Ejecutando hilo: Hilo 1Contador: 7
Ejecutando hilo: Hilo 1Contador: 8
Ejecutando hilo: Hilo 1Contador: 9
Ejecutando hilo: Hilo 2Contador: 0
Ejecutando hilo: Hilo 2Contador: 1
Ejecutando hilo: Hilo 2Contador: 2
Ejecutando hilo: Hilo 2Contador: 3
Ejecutando hilo: Hilo 2Contador: 4
Ejecutando hilo: Hilo 2Contador: 5
Ejecutando hilo: Hilo 2Contador: 6
Ejecutando hilo: Hilo 2Contador: 7
Ejecutando hilo: Hilo 2Contador: 8
Ejecutando hilo: Hilo 2Contador: 9
Ejecutando hilo: Hilo 3Contador: 0
```

.....

## 6. GESTIÓN DE HILOS

- Si “dormimos” cada hilo 1 segundo por iteración añadiendo la instrucción `Thread.sleep(1000);` en el método `run()`, los resultados de la ejecución son diferentes, dando la sensación de que todos los hilos no pasan a la siguiente iteración hasta que no haya finalizado la actual.
- Aunque en el ejemplo se asigna la mayor prioridad al primer hilo, el tema de las prioridades las veremos con más detalle a lo largo del tema.



```
run:
Ejecutando hilo: Hilo 1Contador: 0
Ejecutando hilo: Hilo 2Contador: 0
Ejecutando hilo: Hilo 4Contador: 0
Ejecutando hilo: Hilo 3Contador: 0
Ejecutando hilo: Hilo 5Contador: 0
Ejecutando hilo: Hilo 1Contador: 1
Ejecutando hilo: Hilo 2Contador: 1
Ejecutando hilo: Hilo 3Contador: 1
Ejecutando hilo: Hilo 4Contador: 1
Ejecutando hilo: Hilo 5Contador: 1
Ejecutando hilo: Hilo 1Contador: 2
Ejecutando hilo: Hilo 2Contador: 2
Ejecutando hilo: Hilo 3Contador: 2
Ejecutando hilo: Hilo 5Contador: 2
Ejecutando hilo: Hilo 4Contador: 2
Ejecutando hilo: Hilo 1Contador: 3
Ejecutando hilo: Hilo 2Contador: 3
Ejecutando hilo: Hilo 3Contador: 3
Ejecutando hilo: Hilo 4Contador: 3
Ejecutando hilo: Hilo 5Contador: 3
```

.....

## 7. SINCRONIZACIÓN DE HILOS

- En un entorno multiproceso, varios subprocesos podrían intentar modificar el mismo recurso.
- Por ejemplo, supongamos que en 2 o más bancos hacen transacciones sobre una misma cuenta. Si en uno de los bancos (un hilo) cambia el valor de la cuenta mientras que en otro banco se realiza una transacción sobre esa misma cuenta se podrían dar situaciones desagradables, como por ejemplo que el saldo final de la cuenta no fuera el que tiene que ser.
- Para evitar esto habría que impedir que 2 o más hilos (del mismo o diferentes bancos) accedieran simultáneamente al saldo de la cuenta. Es decir, los hilos tienen que sincronizarse para hacer las operaciones de forma "ordenada".
- En programación, para evitar este tipo de problemas tenemos que proteger el recurso compartido, indicando que es una **sección crítica** utilizando la palabra reservada **synchronized**.
- Una forma de coordinar accesos a secciones críticas por varios subprocesos es a través de las acciones **esperar** y **notificar**.

## 7. SINCRONIZACIÓN DE HILOS

### Esperar

- Método **wait()**.
- Este método hace que el subproceso actual espere "indefinidamente" hasta que otro subproceso le de paso (despierte).
- Hablando de estados de un hilo, este método "bloquea" un hilo.

### Notificar

- Esta acción "despierta" hilos que están esperando.
- Hay dos formas de hacerlo:
- Método **notify()**:
  - ✓ Para todos los subprocesos que esperan "en el monitor de este objeto", este método despierta a cualquiera de ellos.
  - ✓ La elección de exactamente qué hilo despertar no es determinista y depende de la implementación.

## 7. SINCRONIZACIÓN DE HILOS

### Esperar



### Notificar

- Método `notifyAll()`:
  - ✓ Este método activa (despierta) todos los hilos que están esperando en el monitor de este objeto.
  - ✓ Los hilos despertados se completarán de la manera habitual, como cualquier otro hilo.

## 7. SINCRONIZACIÓN DE HILOS

- Veamos un sencillo ejemplo para entender la necesidad de sincronizar. Tenemos la siguiente clase Contador, que es compartida por 2 hilos:

```
public class Contador {  
  
    private int cont;  
  
    public Contador(int cont) {  
        this.cont = cont;  
    }  
  
    public int getCont() {  
        return cont;  
    }  
  
    public void incrementar() {  
        this.cont++;  
    }  
  
    public void decrementar() {  
        this.cont--;  
    }  
  
}
```

## 7. SINCRONIZACIÓN DE HILOS

- Los hilos son los siguientes:

```
public class HiloA implements Runnable {  
  
    private Contador c;  
  
    public HiloA(Contador c) {  
        this.c = c;  
    }  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            c.incrementar();  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException ex) {  
                ex.printStackTrace();  
            }  
            System.out.println(this.getClass().getName() +  
                               " contador: " + c.getCont());  
        }  
    }  
}
```

```
public class HiloB implements Runnable {  
  
    private Contador c;  
  
    public HiloB(Contador c) {  
        this.c = c;  
    }  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            c.decrementar();  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException ex) {  
                ex.printStackTrace();  
            }  
            System.out.println(this.getClass().getName() +  
                               " contador: " + c.getCont());  
        }  
    }  
}
```





## 7. SINCRONIZACIÓN DE HILOS

- Como vemos los dos hilos son idénticos, salvo que HiloA llama al método que incrementa el contador, y HiloB al que lo decrementa. Además con la instrucción *Thread.sleep(100);* dormimos a los hilos en cada iteración para que se vayan alternando en su ejecución, y evitar que se ejecute por completo uno y después el otro.
- Para probar el código realizamos lo siguiente:

```
public class AplicacionContador {  
  
    public static void main(String[] args) {  
        Contador c=new Contador(100);  
        Thread hiloA=new Thread(new HiloA(c));  
        Thread hiloB=new Thread(new HiloB(c));  
        hiloA.start();  
        hiloB.start();  
    }  
}
```

## 7. SINCRONIZACIÓN DE HILOS

- Antes de ver los resultados de la ejecución, podríamos esperar los siguientes resultados finales del contador:  
    HiloA contador: 200   ya que  empieza en 100 y le suma 100  
    HiloB contador: 100   ya que  empieza en 200 y le resta 100
- Pero vemos que los resultados son bien distintos, variando incluso de una ejecución a otra.
- Esto es debido, a que el acceso al contador por ambos hilos se hace de forma totalmente "desordenada". A lo mejor mientras HiloA está incrementando, antes de realizar la operación, HiloB decrementa, y viceversa.
- Si queremos que esto no ocurra y que los resultados sean los esperados, deberíamos asegurarnos que las operaciones sobre el contador se realizan de forma atómica, es decir, hasta que no se incrementa no se permite el decremento y viceversa.

## 7. SINCRONIZACIÓN DE HILOS

- Es decir, tenemos que indicar que las zonas del código donde cada hilo modifica el contador son regiones críticas con la palabra reservada `synchronized`:

```
@Override
public void run() {
    synchronized (c) {
        for (int i = 0; i < 100; i++) {
            c.decrementar();
            try {
                Thread.sleep(100);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
            System.out.println(this.getClass().getName()
                               + " contador: " + c.getCont());
        }
    }
}
```

```
@Override
public void run() {
    synchronized (c) {
        for (int i = 0; i < 100; i++) {
            c.incrementar();
            try {
                Thread.sleep(100);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
            System.out.println(this.getClass().getName()
                               + " contador: " + c.getCont());
        }
    }
}
```

## 7. SINCRONIZACIÓN DE HILOS

- Como vemos, la región crítica delimitada por el bloque `synchronized`, recibe como parámetro cuál es el recurso compartido a sincronizar.
- De esta forma, cuando un hilo intenta acceder a una región crítica, pregunta al contador si hay algún otro hilo que esté accediendo a él.  
Si es que sí, entonces el hilo queda bloqueado “temporalmente” hasta que se libere el contador, momento en el que accede.  
Si el contador está libre, entonces el hilo entra en su región crítica bloqueando al contador, ejecuta su bloque de código y hasta que no salga de él no desbloquea al contador.
- El desbloqueo del contador se producirá cuando el hilo que lo está bloqueando sale de su región crítica porque finaliza normalmente el código o porque se produce una excepción.
- Ahora si ejecutamos, obtenemos los resultados esperados.



## 8. MÉTODOS SINCRONIZADOS

- Para una mejor implementación de la sincronización de hilos, se debe evitar la sincronización de bloques de código, y sustituirlas por la **sincronización de métodos**.
- Para ello simplemente añadimos la palabra **synchronized** en la declaración del método.
- Así la clase **Contador** del ejemplo anterior quedaría así:

```
public class Contador {  
  
    private int cont;  
  
    public Contador(int cont) {  
        this.cont = cont;  
    }  
  
    synchronized public int getCont() {  
        return cont;  
    }  
  
    synchronized public void incrementar() {  
        this.cont++;  
    }  
  
    synchronized public void decrementar() {  
        this.cont--;  
    }  
  
}
```

## 8. MÉTODOS SINCRONIZADOS

- Veamos otro ejemplo de sincronización.
- Se dispone de la clase **Cuenta** que modela una cuenta bancaria de la que es posible retirar dinero siempre que haya saldo.
- Varias personas pueden ser titulares de la misma cuenta y entonces sacar dinero de ella, en cualquier momento.
- Puede darse el caso de que 2 personas estén intentando sacar dinero a la vez en distintos cajeros, y entonces es necesario realizar una sincronización de la operación, para no obtener resultados desagradables como pudiera ser el siguiente:
  - ✓ En la cuenta hay un saldo de 50€.
  - ✓ A las 12:00 María está sacando 50€.
  - ✓ Llega en ese momento Luis a otro cajero para sacar 50€.
  - ✓ Los 2 acceden a la cuenta en el mismo momento y como hay saldo, retiran un total de 100€, cuando en realidad sólo había 50€.
  - ✓ Resultado: números rojos (-50€).

## 8. MÉTODOS SINCRONIZADOS

- Solución: sincronización.

```
public class Cuenta {  
  
    private double saldo;  
  
    public Cuenta(double saldo) {  
        this.saldo = saldo;  
    }  
  
    public double getSaldo() {  
        return saldo;  
    }  
  
    synchronized public void retirar(double cantidad) {  
        if (this.saldo >= cantidad) {  
            System.out.println("Saldo actual: " + this.saldo);  
            try {  
                System.out.println("Retirando....");  
                Thread.sleep(1000);  
            } catch (InterruptedException ex) {  
                ex.printStackTrace();  
            }  
            this.saldo = this.saldo - cantidad;  
            System.out.println("Saldo despues de retirar: " + this.saldo);  
        } else {  
            System.out.println("No hay saldo disponible");  
        }  
    }  
}
```

```
public class Titular implements Runnable {  
  
    private Cuenta cta;  
    private String nombreTitular;  
  
    public Titular(Cuenta cta, String nombreTitular) {  
        this.cta = cta;  
        this.nombreTitular = nombreTitular;  
    }  
  
    @Override  
    public void run() {  
        cta.retirar(50);  
    }  
}
```

## 8. MÉTODOS SINCRONIZADOS

- Solución: sincronización.

```
public class AplicacionCuenta {  
    public static void main(String[] args) {  
        Cuenta cuenta=new Cuenta(50);  
        Titular maria=new Titular(cuenta,"María");  
        Titular luis=new Titular(cuenta,"Luis");  
        Thread hilo1=new Thread(maria);  
        Thread hilo2=new Thread(luis);  
        hilo1.start();  
        hilo2.start();  
    }  
}
```



## 8. MÉTODOS SINCRONIZADOS

### Resultado sin sincronización

#### Output - Procesos (run)

```
run:
Saldo actual: 50.0
Retirando....
Saldo actual: 50.0
Retirando....
Saldo despues de retirar: 0.0
Saldo despues de retirar: -50.0
BUILD SUCCESSFUL (total time: 1 second)
```

### Resultado con sincronización

#### Output - Procesos (run)

```
run:
Saldo actual: 50.0
Retirando....
Saldo despues de retirar: 0.0
No hay saldo disponible
BUILD SUCCESSFUL (total time: 1 second)
```

## 9. MODELO PRODUCTOR-CONSUMIDOR

- A estas alturas, podemos determinar que una de las ventajas que tiene la programación multihilo, es que los hilos pueden comunicarse entre sí.  
De tal forma, que los hilos pueden utilizar objetos comunes a todos ellos, pero que cada hilo manipula independientemente.
- Uno de los ejemplos clásicos de comunicación entre hilos es el **modelo productor-consumidor**.
- En este modelo, uno hilo o más hilos producen datos, que otro u otros hilos usan (consumen).
- Gráficamente tendríamos lo siguiente:



## 9. MODELO PRODUCTOR-CONSUMIDOR

- El **monitor** será el objeto encargado de sincronizar los hilos. Podemos verlo como una tubería que conecta al productor y al consumidor.
- El problema puede venir por 2 motivos:
  - a) El hilo productor produce datos más rápido que el hilo consumidor los consume, provocando que el consumidor se "salte" algún dato.
  - b) El hilo consumidor consume datos más rápido que el productor los produce, provocando que el consumidor repita el mismo dato, o que no tenga datos que consumir.
- Para evitar este problema, es necesario que los hilos se sincronicen, y que cuando un productor esté generando un dato lo "notifique" para que los consumidores "esperen", y al revés, que cuando un consumidor esté consumiendo un dato lo "notifique" para el productor "espere" a generar el siguiente.
- En este proceso intervienen entonces los métodos:
  - ✓ `wait()`
  - ✓ `notify()` / `notifyAll()`

## 9. MODELO PRODUCTOR-CONSUMIDOR

- Veámoslo con un ejemplo.
- Un hilo productor va ir produciendo números de uno en uno, y una serie de consumidores irán consumiendo dichos números. El productor no podrá producir más de un número y los consumidores no podrán consumir nada si no se ha producido antes.
- Creamos el contenedor común donde el productor dejará los datos y los consumidores irán cogiendo de él. Además será el encargado de sincronizar a productor y consumidores, es decir esta clase será la que actúa de **monitor**:

## 9. MODELO PRODUCTOR-CONSUMIDOR

```
public class Contenedor {  
  
    private int dato;  
    private boolean hayDato = false;  
  
    public synchronized int get() {  
        while (!hayDato) {  
            try {  
                wait(); // Consumidor tiene que esperar  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
        // Ya hay dato, entonces lo consume  
        hayDato = false;  
        // Y notifica al productor que ya lo ha consumido  
        notify();  
        return dato;  
    }  
}
```

```
public synchronized void put(int value) {  
    while (hayDato) {  
        try {  
            wait(); // Productor tiene que esperar  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
    // Ya no hay dato, entonces lo produce  
    dato = value;  
    hayDato = true;  
    // Y notifica que lo ha producido  
    notify();  
}
```

## 9. MODELO PRODUCTOR-CONSUMIDOR

- A continuación, diseñamos el productor y el consumidor:

```
public class Productor implements Runnable {  
  
    private Random aleatorio;  
    private Contenedor contenedor;  
    private int idproductor;  
    private int TIEMPOESPERA = 1000;  
  
    public Productor(Contenedor contenedor, int idproductor) {  
        this.contenedor = contenedor;  
        this.idproductor = idproductor;  
        aleatorio = new Random();  
    }  
  
    public void run() {  
        while (Boolean.TRUE) {  
            int valor = aleatorio.nextInt(100);  
            contenedor.put(valor);  
            System.out.println("El productor " + idproductor + " pone: " + valor);  
            try {  
                Thread.sleep(TIEMPOESPERA);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```
public class Consumidor implements Runnable {  
  
    private Contenedor contenedor;  
    private int idconsumidor;  
  
    public Consumidor(Contenedor contenedor, int idconsumidor) {  
        this.contenedor = contenedor;  
        this.idconsumidor = idconsumidor;  
    }  
  
    public void run() {  
        while (Boolean.TRUE) {  
            System.out.println("El consumidor " + idconsumidor + " consume: " + contenedor.get());  
        }  
    }  
}
```

## 9. MODELO PRODUCTOR-CONSUMIDOR

- Finalmente programamos una aplicación que arranque los hilos. Esta aplicación estará corriendo hasta que nosotros la paremos:

```
public class Aplicacion {  
  
    private static Contenedor contenedor;  
    private static Thread productor;  
    private static Thread[] consumidores;  
    private static final int NUMCONSUMIDORES = 5;  
  
    public static void main(String[] args) {  
        contenedor = new Contenedor();  
        productor = new Thread(new Productor(contenedor, 1));  
        consumidores = new Thread[NUMCONSUMIDORES];  
  
        for (int i = 0; i < NUMCONSUMIDORES; i++) {  
            consumidores[i] = new Thread(new Consumidor(contenedor, i));  
            consumidores[i].start();  
        }  
  
        productor.start();  
    }  
}
```

Creamos un productor y 5 consumidores

## 10. GESTIÓN DE PRIORIDADES

- Java tiene un scheduler que monitoriza todos los hilos que se están ejecutando en todos los programas y decide cuales deben ejecutarse.
- Para tomar la decisión comprueba cual es la prioridad del hilo, de tal forma, que aquellos con prioridad más alta dispondrán de la CPU antes que los que tienen prioridad más baja.
- El rango de prioridades oscila entre 1 y 10.
- La prioridad por defecto de un hilo es NORM\_PRIORITY, que tiene asignado un valor de 5.
- Podemos asignar distintas prioridades a los hilos usando las variables estáticas MIN\_PRIORITY (1) y MAX\_PRIORITY (10) usando el método **setPriority(valor)**.
- En realidad el sistema operativo no está obligado a respetar las prioridades, sino que se lo tomará como recomendaciones.
- En la práctica, casi nunca habrá que establecer las prioridades a mano.



## 10. GESTIÓN DE PRIORIDADES

- Vamos a hacer un programa que cuenta hasta 1000 entre tres hilos mostrando al finalizar un resumen de las veces que ha contado cada uno.
- Cada hilo tendrá una prioridad aleatoria entre 1 y 10.
- El contador es el recurso compartido:

```
public class Contador {  
  
    private int contador=0;  
  
    synchronized public int getContador() {  
        return contador;  
    }  
  
    synchronized public void incrementarContador() {  
        this.contador++;  
    }  
}
```

## 10. GESTIÓN DE PRIORIDADES

```
public class Hilo implements Runnable {  
  
    private int id; //id del hilo de ejecución  
    private Contador contador;  
    private int veces; // veces que se ha utilizado cada hilo  
    private int prioridad;  
  
    public Hilo(int id, int prioridad, Contador c) {  
        this.contador = c;  
        this.id = id;  
        this.prioridad=prioridad;  
    }  
  
    @Override  
    public void run() {  
        while (contador.getContador() < 1000) {  
            contador.incrementarContador();  
            veces++;  
            System.out.println("Hilo: " + id + " (" + this.prioridad + ") "  
                               + " incrementa contador" + contador.getContador()  
                               + "--> Veces: " + this.veces);  
        }  
    }  
}
```

## 10. GESTIÓN DE PRIORIDADES

```
public class Aplicacion {  
  
    public static void main(String[] args) {  
  
        Contador contador = new Contador();  
        Hilo[] hilos = new Hilo[3];  
  
        for (int i = 0; i < 3; i++) {  
            int prioridad=(int) ((10*Math.random())+1);  
            Hilo hilo = new Hilo(i + 1,prioridad, contador);  
            Thread hebra = new Thread(hilo);  
            hebra.setPriority(prioridad);  
            hebra.start();  
        }  
    }  
}
```

- Varias ejecuciones del programa generarán salidas diferentes.
- El hilo con mayor prioridad será “normalmente” el que más veces cuente, pero no tiene por qué.
- El comportamiento no está garantizado.