UD 2. PROGRAMACIÓN DE HILOS

Programación de servicios y procesos

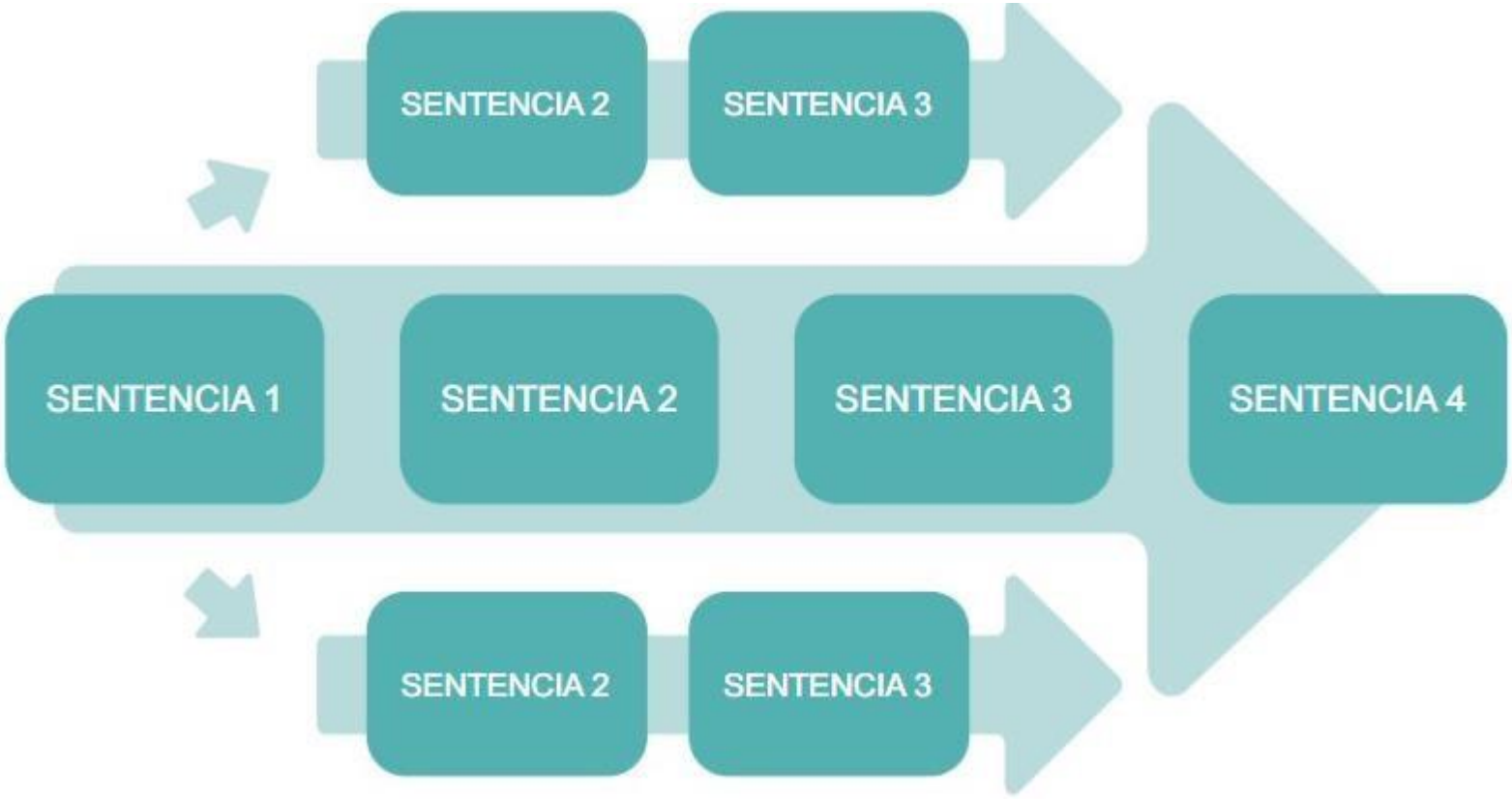
1. Conceptos básicos

Desde el punto de vista del sistema operativo, un programa informático en ejecución es un proceso que compite con los demás procesos por acceder a los recursos del sistema. Visto desde dentro, un programa es, básicamente una sucesión de sentencias que se ejecutan una detrás de otra.

En el caso de un programa absolutamente secuencial, el hilo de ejecución es único, lo que provoca que cada sentencia tenga que esperar que la sentencia inmediatamente anterior se ejecute completamente antes de comenzar su ejecución, es decir, las instrucciones se van ejecutando una detrás de otra y no de manera secuencial.

En cambio, en un programa multihilo, algunas de las sentencias se ejecutan simultáneamente, ya que los hilos creados y activos en un momento dado acceden a los recursos de procesamiento sin necesitar esperar a que otras partes del programa terminen.

1. Conceptos básicos



1. Conceptos básicos

Algunas de las características que tienen los hilos son:

- **Dependencia del proceso**. Siempre se ejecutan dentro del contexto de un proceso.
- **Ligereza**. Al ejecutarse dentro del contexto de un proceso, no requiere generar procesos nuevos, por lo que son óptimos desde el punto de vista del uso de recursos.
- **Compartición de recursos**. Dentro del mismo proceso, los hilos comparten espacio de memoria. Esto implica que pueden sufrir colisiones en los accesos a las variables provocando errores de concurrencia.
- **Paralelismo**. Aprovechan los núcleos del procesador generando un paralelismo real, siempre dentro de las capacidades del procesador.
- **Concurrencia.** Permiten atender de manera concurrente múltiples peticiones. Esto es especialmente importante en servidores web y de bases de datos, por ejemplo.

Para ver más claro cómo se comporta un programa de un único hilo frente a los programas de múltiples hilos, se puede establecer la siguiente analogía.

El camarero de una cafetería recibe a un cliente que le pide un café, una tostada y una tortilla francesa. Si el camarero trabaja como un proceso de un único hilo, pondrá la cafetera a preparar café, esperará a que termine para poner el pan a tostar y no pedirá la tortilla a la cocina hasta que el pan no esté tostado.

La mayoría de los camareros suelen trabajar como procesos multihilo: ponen la cafetera a preparar el café, el pan a tostar y piden a la cocina los platos sin esperar a que cada una de las demás tareas esté terminada. Dado que cada una de ellas consume

1. Conceptos básicos

diferentes recursos, no es necesario hacerlo. Transcurrido el tiempo que tarda en estar lista la tarea más lenta, el cliente estará atendido.

Siguiendo con la analogía, al igual que pasa en los ordenadores, los recursos de una cafetería son limitados, por lo que hay una restricción física que impide que se hagan más tareas simultáneamente de las que permiten los recursos. Si solo hay una tostadora con capacidad para dos rebanadas de pan, no se podrán tostar más de dicho número en un mismo instante de tiempo.

La programación que permite realizar este tipo de sistemas se llama multihilo, concurrente o asíncrona.

Ejemplo de programa que se ejecuta en un hilo único.

Raton.java

La salida producida es la siguiente:

```
El ratón Fievel ha comenzado a alimentarse
El ratón Fievel ha terminado de alimentarse
El ratón Jerry ha comenzado a alimentarse
El ratón Jerry ha terminado de alimentarse
El ratón Pinky ha comenzado a alimentarse
El ratón Pinky ha terminado de alimentarse
El ratón Mickey ha comenzado a alimentarse
El ratón Mickey ha terminado de alimentarse
```

1. Conceptos básicos

Este tipo de ejecución se conoce como secuencial o no concurrente: cada sentencia debe esperar a que se ejecute la sentencia anterior. Como consecuencia, cada ratón tardará en comenzar a comer tanto tiempo como tarden los ratones que comenzaron antes que él. El tiempo total del proceso es, por tanto, la suma de todos los tiempos parciales (en este caso, 18 segundos).

2. Programación concurrente o multihilo

El ejemplo anterior se puede programar de manera concurrente de forma sencilla, ya que no hay ningún recurso compartido. Para ello, basta con convertir cada objeto de la clase Raton en un hilo (thread) y programar aquello que se quiere que ocurra concurrentemente dentro del método run. Una vez creadas las instancias, se invoca al método start de cada una de ellas, lo que provoca la ejecución del contenido del método run en un hilo independiente.

RatonMH.java

La salida producida es la siguiente:

```
El ratón Jerry ha comenzado a alimentarse
El ratón Mickey ha comenzado a alimentarse
El ratón Fievel ha comenzado a alimentarse
El ratón Pinky ha comenzado a alimentarse
El ratón Pinky ha terminado a alimentarse
El ratón Fievel ha terminado a alimentarse
El ratón Jerry ha terminado a alimentarse
El ratón Mickey ha terminado a alimentarse
```

Todos los ratones han comenzado a alimentarse de inmediato, sin esperar a que termine ninguno de los demás. El tiempo total del proceso será, aproximadamente, el tiempo del proceso más lente (en este caso, 6 segundos).

2. Programación concurrente o multihilo

En Java existen dos formas para la creación de hilos:

- Implementando la interface *java.lang.Runnable*.
- Heredando de la clase *java.lang.Thread*.

La implementación de la interface *Runnable* obliga a programar el método sin argumentos run.

```
public class HiloViaInterface implements Runnable{
        public void run(){
        }
}
```

Heredando de la clase *Thread* esto no es obligatorio porque dicha clase ya es una implementación de la interface *Runnable*.

No obstante, crear un hilo heredando de Thread "necesita" la implementación del método sin argumentos run, ya que de lo contrario la clase no tendrá un comportamiento multihilo.

Retomando el enunciado del ejemplo de los objetos de la clase RatonMH, la solución mediante implementación de la interface Runnable tendría el código que se muestra a continuación:

2. Programación concurrente o multihilo

Errores de concurrencia

En el siguiente ejemplo, se implementa un hilo mediante la interface Runnable para crear múltiples hilos a partir de un único objeto.

HiloSimple.java

El resultado sería:

```
El ratón Fievel ha comenzado a alimentarse
El ratón Fievel ha comenzado a alimentarse
El ratón Fievel ha comenzado a alimentarse
El ratón Fievel ha comenzado a alimentarse
El ratón Fievel ha terminado a alimentarse
El ratón Fievel ha terminado a alimentarse
Alimento consumido: $d
El ratón Fievel ha terminado a alimentarse
Alimento consumido: $d
El ratón Fievel ha terminado a alimentarse
Alimento consumido: $d
Alimento consumido: $d
```

Cada hilo ha ejecutado el método *run* sobre los datos del mismo objeto. Es decir, se ha ejecutado simultáneamente cuatro veces un bloque de código de un único objeto, compartiendo sus atributos. De esta forma, en la salida se puede apreciar que el valor del atributo alimentoConsumido se ha incrementado en 1 por cada hilo. Esto se puede observar porque el valor de *alimentoConsumido* se ha incrementado varias veces durante la ejecución.

2. Programación concurrente o multihilo

Errores de concurrencia

Por ejemplo, sustituyendo el método *main* del ejemplo anterior por el siguiente código, se pueden crear y ejecutar varios *threads* mediante un bucle *for* a partir de una única instancia de una clase que implementa la interfaz *Runnable*.

```
public static void main(String[] args){
        RatonSimple fievel = new RatonSimple("Fievel", 4);
        for(int i=0; i<100; i++){ new
        Thread(fievel).start();
        }
}
```

El resultado, con un número bajo de iteraciones será habitualmente correcto, pero con un número alto, el resultado normalmente será erróneo, ya que el atributo *alimentoConsumido* tomará un valor por debajo del número de iteraciones.

Esto se debe a que al compartir todos los hilos los mismos atributos producen errores de concurrencia que deben evitarse mediante técnicas concretas.

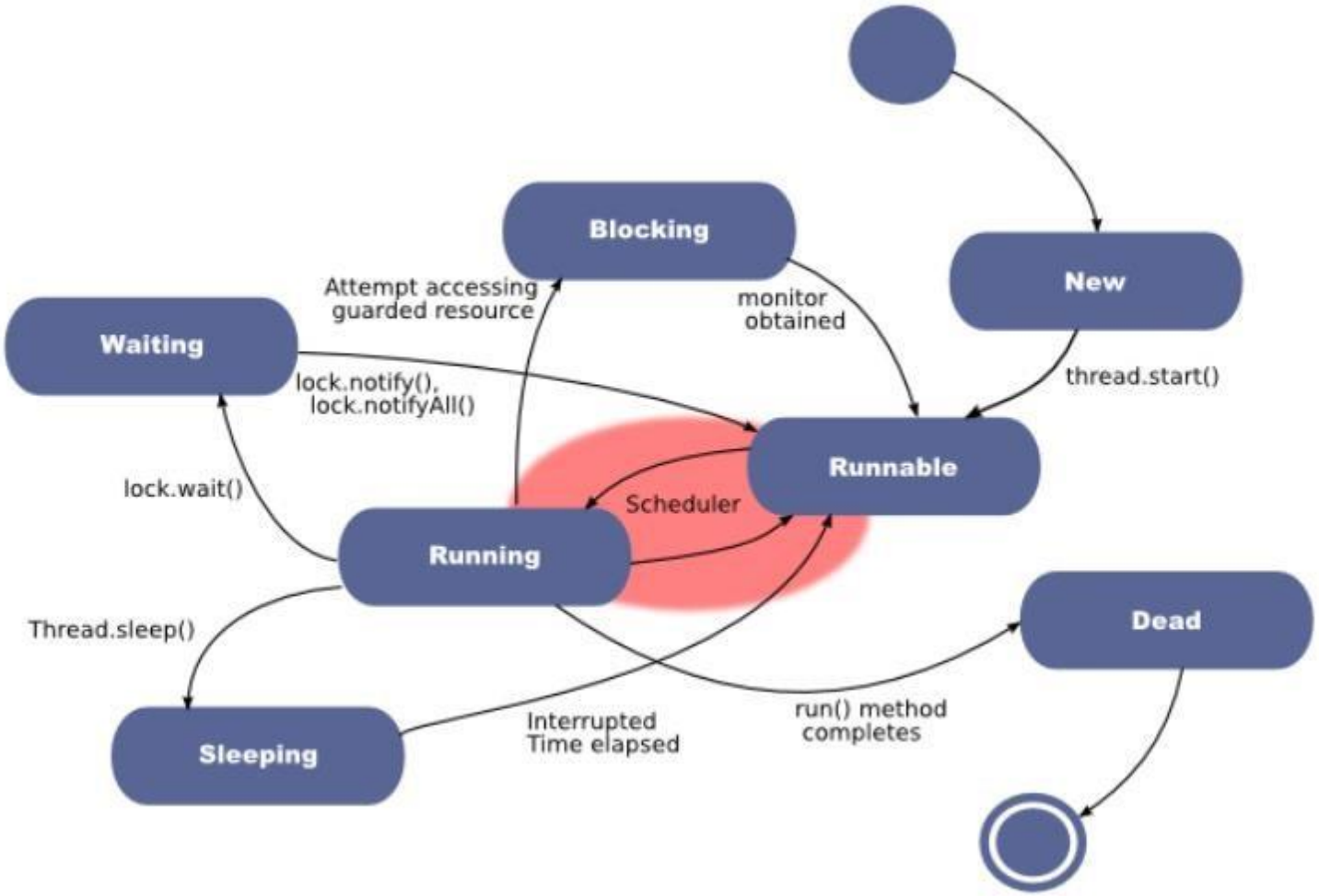
3. Estados de un hilo

Los hilos pueden cambiar de estado a lo largo de su ejecución. En Java, están recogidos dentro de la enumeración *State* contenida dentro de la clase *java.lang.Thread*.

El estudio de un hilo se obtiene mediante el método *getState()* de la clase *Thread*, que devolverá algunos de los valores posibles recogidos en la enumeración indicada anteriormente.

Estado	Valor en Thread.State	Descripción
Nuevo	NEW	El hilo está creado, pero aún no se ha arrancado.
Ejecutable	RUNNABLE	El hilo está arrancado y podría estar en ejecución o pendiente de ejecución.
Bloqueado	BLOCKED	Bloqueado por un monitor.
Esperando	WAITING	El hilo está esperando a que otro hilo realice una acción determinada.
Esperando un tiempo	TIME_WAITING	El hilo está esperando a que otro hilo realice una acción determinada en un período de tiempo concreto.
Finalizado	TERMINATED	El hilo ha terminado su ejecución.

3. Estados de un hilo

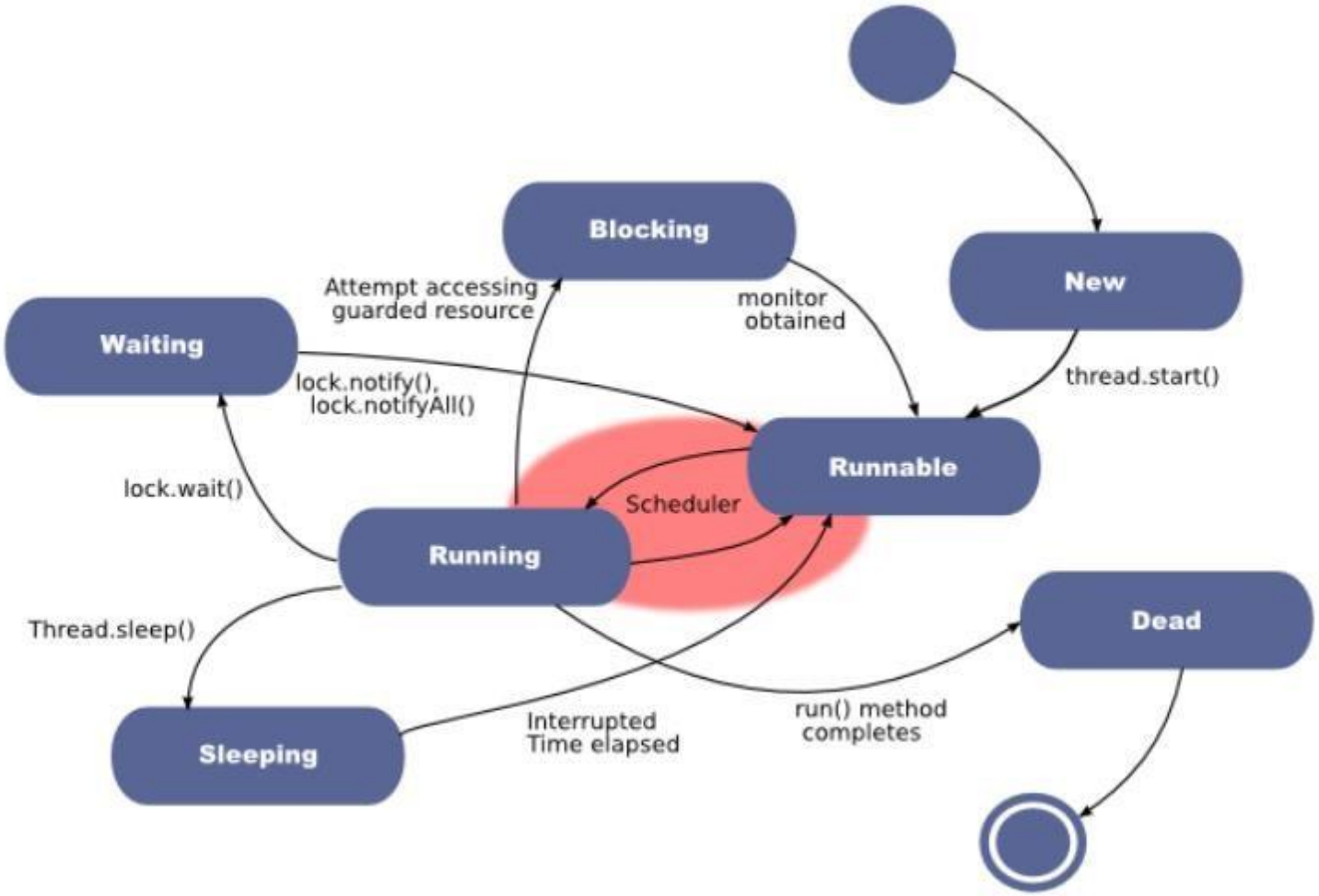


3. Estados de un hilo

En el siguiente código de ejemplo, se almacenan en un ArrayList los estados por los que pasa un hilo que contiene en su interior una llamada al método sleep. Se utiliza la clase Raton que implmenta Runnable de los ejemplos anteriores.

```
public static void main(String[] args) {
    Raton mickey=new Raton("Mickey, 6");
    ArrayList<Thread.State> estadosHilo=new
    ArrayList();
    Thread h=new Thread(mickey);
    estadosHilo.add(h.getState());
    h.start();
    while(h.getClass()!=Thread.State.TERMINATED) {
        if(!estadosHilo.contains(h.getState())) {
            estadosHilo.add(h.getState());
        }
    }
    if(!estadosHilo.contains(h.getClass())) {
        estadosHilo.add(h.getState());
    }
    for(Thread.State estado : estadosHilo) {
        System.out.println(estado);
    }
}
```

3. Estados de un hilo



4. Gestión de hilos

Operaciones básicas:

- **Creación y arranque de hilos**. Cualquier programa a ejecutarse es un proceso que tiene un hilo de ejecución principal. Este hilo puede a su vez crear nuevos hilos que ejecutarán código diferente o tareas, es decir el camino de ejecución no tiene por qué ser el mismo.
    - En Java existen dos formas para crear hilos: extendiendo la clase **Thread** o implementando la interfaz **Runnable.** Ambas son parte del paquete **java.lang.**
- **Espera de hilos**. Como varios hilos comparten el mismo procesador, si alguno no tiene trabajo que hacer, es bueno suspender su ejecución para que haya un mayor tiempo de procesador disponible.

4.1. Gestión de hilos. Operaciones básicas en Java

CREACIÓN DE HILOS. **LA CLASE THREAD**

- La forma más simple de añadir funcionalidad de hilo a una clase es extender la clase **Thread**. O lo que es lo mismo, crear una subclase de la clase **Thread**. Esta subclase debe sobrescribir el método **run()** con las acciones que el hilo debe desarrollar.

```
class NombreHilo extends Thread{
    //propiedades, constructores y metodos de la clase
    public void run() {
    //acciones que lleva a cabo el hilo
    }
}
```

La clase Thread representa un hilo de ejecución. Cuando una clase hereda de *Thread* puede implementar el método *run* y ejecutarse de forma asíncrona.

Para lanzar un objeto *Thread* de manera asíncrona, basta ejecutar su método *start*.

4.1. Gestión de hilos. Operaciones básicas en Java

CREACIÓN DE HILOS. **LA CLASE THREAD**

Como hemos dicho, En Java para utilizar la multitarea debemos de usar la clase Thread (es decir que la clase que implementemos debe heredar de la clase Thread) y la clase Thread implementa la Interface Runnable. En el siguiente diagrama de clase mostramos la Interface Runnable y la clase Thread con sus principales métodos:

4.1. Gestión de hilos. Operaciones básicas en Java (Resumen de la clase Thread)

CREACIÓN DE HILOS. **LA CLASE THREAD**

Como hemos dicho, En Java para utilizar la multitarea debemos de usar la clase Thread (es decir que la clase que implementemos debe heredar de la clase Thread) y la clase Thread implementa la Interface Runnable. En el siguiente diagrama de clase mostramos la Interface Runnable y la clase Thread con sus principales métodos:

Método	Tipo de retorno	Descripción
start()	void	Implementa la operación *create*. Comienza la ejecución del hilo de la clase correspondiente. Llama al método *run()*
run()	void	Si el hilo se construyó implementando la interfaz *Runnable*, entonces se ejecuta el método *run()* de ese objeto. En caso contrario, no hace nada.
currentThread()	static Thread	Devuelve una referencia al objeto hijo que se está ejecutando actualmente
join()	void	Implementa la operación *join* para hilos.
sleep(long milis)	static void	El hilo que se está ejecutando se suspende durante el número de milisegundos especificado
interrupt()	void	Interrumpe el hilo del objeto
interrupted()	static boolean	Comprueba si el hilo ha sido interrumpido
isAlive()	boolean	Devuelve *true* en caso de que el hilo esté vivo, es decir, no hay terminado el método *run()* en su ejecución.

4.2. Gestión de hilos. Creación y arranque de hilos

Extendiendo de la clase Thread:

```
class HelloThread extends Thread {
    public void run() {
        System.out.println("Hola desde el hilo creado!");
    }
    } public class RunThreadET { public static void main(String args[]) { new
HelloThread().start();// Crea y arranca un nuevo hilo de ejecución
        System.out.println("Hola desde el hilo principal!");
        System.out.println("Proceso acabando.");
    }
}
```

4.2. Gestión de hilos. Creación y arranque de hilos

Extendiendo de la clase Thread:

```
class HelloThread extends Thread {
    public void run() {
        System.out.println("Hola desde el hilo creado!");
    }
    } public class RunThreadET { public static void main(String args[]) { new
HelloThread().start();// Crea y arranca un nuevo hilo de ejecución
        System.out.println("Hola desde el hilo principal!");
        System.out.println("Proceso acabando.");
    }
}
```

4.2. Gestión de hilos. Creación y arranque de hilos

Otro ejemplo extendiendo de la clase Thread:

```
public class Tarea extends Thread {
  @Override public void run() {
  for (int i = 0; i < 10; i++) {
      System.out.println("Soy un hilo y esto es lo que hago");
    }
  } } public class Programa { public
static void main(String args[]) { Tarea
tarea = new Tarea(); tarea.start();
    System.out.println("Yo soy el hilo principal y sigo haciendo mi trabajo");
    System.out.println("Fin del hilo principal");
  }
}
```

4.1. Gestión de hilos. Operaciones básicas en Java

Como hemos dicho, En Java para utilizar la multitarea debemos de usar la clase Thread (es decir que la clase que implementemos debe heredar de la clase Thread) y la clase Thread implementa la Interface Runnable. En el siguiente diagrama de clase mostramos la Interface Runnable y la clase Thread con sus principales métodos:

4.1. Gestión de hilos. Operaciones básicas en Java (Resumen de la clase Thread)

Método	Tipo de retorno	Descripción
start()	void	Implementa la operación *create*. Comienza la ejecución del hilo de la clase correspondiente. Llama al método *run()*
run()	void	Si el hilo se construyó implementando la interfaz *Runnable*, entonces se ejecuta el método *run()* de ese objeto. En caso contrario, no hace nada.
currentThread()	static Thread	Devuelve una referencia al objeto hijo que se está ejecutando actualmente
join()	void	Implementa la operación *join* para hilos.
sleep(long milis)	static void	El hilo que se está ejecutando se suspende durante el número de milisegundos especificado
interrupt()	void	Interrumpe el hilo del objeto
interrupted()	static boolean	Comprueba si el hilo ha sido interrumpido
isAlive()	boolean	Devuelve *true* en caso de que el hilo esté vivo, es decir, no hay terminado el método *run()* en su ejecución.

4.1. Gestión de hilos. Operaciones básicas en Java

```
package ejemploHilos;

/*El siguiente ejemplo muestra el uso de algunos de los métodos anteriores:*/ public class

HiloEjemplo2 extends Thread {

            public void run() {

                        System.out.println("Dentro del Hilo:" + this.getName () + ", Prioridad: " + this.getPriority() + ", ID: " +
            this.getId() );

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

            HiloEjemplo2 h = null; for (int i = 0; i < 3;

            i++) {

                                    h = new HiloEjemplo2 () ; //crear hilo h

                                    .setName("HIL0"+i); //damos nombre al hilo h

                                    .setPriority(i+1); //damos prioridad

                                    h.start(); //iniciar hilo

                                    System.out.println("Informacion del " + h

                                    .getName() +": "+ h .toString());

                                        System.out.println("3 HILOS CREADOS...");

                        }

            }// main

}// clase
```

4.1. Gestión de hilos. Operaciones básicas en Java

CREACIÓN DE HILOS. **LA INTERFAZ RUNNABLE**

- Para añadir la funcionalidad de hilo a una clase que deriva de otra clase (por ejemplo un applet), siendo esta distinta de Thread, se utiliza la interfaz Runnable. Esta interfaz añade la funcionalidad de hilo a una clase con solo implementarla. Por ejemplo, para añadir la funcionalidad de hilo a un applet definimos la clase como:

  public class Reloj extends Applet implements Runnable {}

- La interfaz Runnable proporciona un Único método, el modo run(). Este es ejecutado por el objeto hilo asociado. La forma general de declarar un hilo implementando la interfaz **Runnable** es la siguiente: **`class NombreHilo implements Runnable {`**

```
    //propiedades, constructores y metodos de la clase
    public void run() {
        //acciones que lleva a cabo el hilo
    }
}
```

Para crear un objeto hilo con el comportamiento de *NombreHilo* escribo: *NombreHilo h =new NombreHilo()*; y para iniciar su ejecución utilizamos el método start(): new Thread(h) .start();

4.2. Gestión de hilos. Creación y arranque de hilos

Implementando la interfaz Runnable:

```
public class RunnableBasico implements Runnable{
        private int id; public
        RunnableBasico(int id) { this.id=id;
        } public void run()
        { while(true) {
                        System.out.println("Procesando hilo "+id);
                        try {
                                Thread.sleep(1000);
                        }catch(InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
```

4.2. Gestión de hilos. Creación y arranque de hilos

Implementando la interfaz Runnable:

```
        public static void main(String[] args) { for(int
                i=0;i<10;i++) { new Thread(new
                RunnableBasico(i)).start();
                }
        }
}
```

4.2. Gestión de hilos. Creación y arranque de hilos

Implementando la interfaz Runnable:

```
class HelloThread implements Runnable {
    Thread t;
    HelloThread () { t = new Thread(this,
        "Nuevo Thread");
        System.out.println("Creado hilo: " + t);
        t.start(); // Arranca el nuevo hilo de ejecución. Ejecuta run
    }
    public void run() {
        System.out.println("Hola desde el hilo creado!");
        System.out.println("Hilo finalizando.");
} }
class RunThreadIR { public static void main(String args[]) {
    new HelloThread(); // Crea un nuevo hilo de ejecución
        System.out.println("Hola desde el hilo principal!");
        System.out.println("Proceso acabando.");
    }
}
```

4.2. Gestión de hilos. Creación y arranque de hilos

De las dos alternativas anteriores, implementar la interfaz Runnable o extender de la clase Thread, ¿cuál utilizar? Depende de la necesidad:

- La utilización de la interfaz Runnable es más general, ya que el objeto puede ser una subclase de una clase distinta de Thread.
- La utilización de la interfaz Runnable no tiene ninguna otra funcionalidad además de run() que la incluida por el programador.
- La extensión de la clase Thread es más fácil de utilizar, ya que está definida una serie de métodos útiles para la administración de hilos. Como comentábamos anteriormente, una clase que hereda de Thread, se convierte automáticamente en un hilo.
- La extensión de la clase Thread está limitada porque los clases creadas como hilos deben ser descendientes únicamente de dicha clase, es decir, una clase que hereda de Thread ya no podrá heredar de ninguna otra, por lo que si la arquitectura de nuestra aplicación lo requiere ya no podríamos.

4.1. Gestión de hilos. Operaciones básicas en Java

- Suspensión de ejecución: el método **sleep**

El método estático sleep de la clase Thread permite suspender la ejecución del hilo desde el que se invoca. Acepta como parámetro la cantidad de tiempo en milisegundos que se desea realizar esta suspensión, cuya precisión queda sometida a la precisión de los temporizadores del sistema y al planificador.

- **Interrupciones**:

En computación, una interrupción es una suspensión temporal de la ejecución de un proceso o de un hilo de ejecución. Las interrupciones suelen pertenecer al sistema operativo, viniendo generadas por peticiones realizadas por dispositivos periféricos.

En Java, una interrupción es una indicación a un hilo de que debe detener su ejecución para hacer otra cosa. Es responsabilidad del programador decidir qué quiere hacer ante una interrupción, siendo lo más habitual detener la ejecución del hilo.

- Una interrupción indica a un hilo que deje hacer lo que esté haciendo (*join*, *sleep*) para hacer otra cosa. Se envía mediante el método ***interrupt*** de la clase *Thread* en el objeto del hilo que se quiere interrumpir.
- El método ***interrupted*** de la clase *Thread* permite saber si un hilo ha sido interrumpido. Reinicia el estado.
- Se capturan mediante la excepción ***InterruptedException***.
- Una vez un hilo ha sido interrumpido se puede finalizar (***return***) o propagar la excepción.
- Mediante el método ***isAlive*()** se puede saber si un thread ha finalizado su ejecución.

4.2. Gestión de hilos. Creación y arranque de hilos

Ejemplo Interrupciones:

```
public class InterrupcionBasico extends Thread{
        public void run() { int contador=0;
        while(true) { contador++; try {
                                System.out.println(contador);
                                if(contador==3) {
                                this.interrupt();
                                }
                                Thread.sleep(1000);
                        }catch(InterruptedException ie){
                                return;
                        }
                }
        }
```

4.2. Gestión de hilos. Creación y arranque de hilos

Ejemplo Interrupciones:

```
        public static void main(String[] args) {
                new InterrupcionBasico().start();
        }
}
```

4.1. Gestión de hilos. Operaciones básicas en Java

- **Compartición de información**

Varios hilos pueden crearse como instancias de la clase *Thread*. Los atributos de dichos hilos, si no son estáticos, serán específicos de cada uno de ellos, por lo que no se podrán utilizar para compartir información.

Si se desea que varios hilos compartan información existen varias alternativas:

- Utilizar atributos estáticos. Estos son comunes a todas las instancias, por lo que independientemente de la manera de construir los hilos la información es compartida.
- Utilizando referencias a objetos comunes accesibles desde todos los hilos. Ejemplo `ComparticionInstaciaUnica.`
- Utilizando atributos no estáticos de la instancia de una clase que implemente Runnable y construyendo los hilos a partir de dicha instancia. Ejemplo `ComparticionRunnable.`

4.2. Gestión de hilos. Creación y arranque de hilos

Ejemplo utilizando referencias comunes desde todos los hilos:

```
public class ComparticionInstaciaUnica extends Thread{ private
        ObjetoComun oc;
        public ComparticionInstaciaUnica(ObjetoComun oc) { this.oc=oc;
        }

        public void run() { this.oc.variableComun++;
                        System.out.println("Variable común: "+this.oc.variableComun);
}
```

4.2. Gestión de hilos. Creación y arranque de hilos

Ejemplo utilizando referencias comunes desde todos los hilos:

```
        public static void main(String[] args) throws InterruptedException{
                ObjetoComun oc=new ObjetoComun();
                ComparticionInstaciaUnica ciu1=new ComparticionInstaciaUnica(oc);
                ComparticionInstaciaUnica ciu2=new ComparticionInstaciaUnica(oc);
                ciu1.start(); Thread.sleep(100); ciu2.start();
        }
}
public class ObjetoComun {
        public int variableComun;
}
```

4.2. Gestión de hilos. Creación y arranque de hilos

Utilizando atributos no estáticos de la instancia de una clase que implemente Runnable :

```
public class ComparticionRunnable implements Runnable{
        private int contador; public void run() {
        contador++;
                System.out.println(contador);
        }
        public static void main(String[] args) throws InterruptedException{
                ComparticionRunnable cb=new ComparticionRunnable(); new
                Thread(cb).start(); Thread.sleep(100); new
                Thread(cb).start();
        }
}
```

4.1. Gestión de hilos. Operaciones básicas en Java

- Espera de hilos ***join***:
  - ***Join*** suspende la ejecución del hilo sobre el que se ejecuta hasta que termina el hilo referenciado en la llamada. Útil cuando algún hilo no tiene nada que hacer o para establecer prioridades. Por ejemplo, si en el hilo actual escribimos hb.join(), el hilo actual se queda en espera hasta que finalice el hilo hb. El método puede generar la excepción InterruptedException que es necesario capturar en un bloque try / catch.
  - Si un Thread necesita esperar a que otro termine (por ejemplo el Thread padre espera a que termine el hijo) puede usar el método join(). ¿Por que se llama así? Crear un proceso es como una bifurcación, se abren 2 caminos, que uno espere a otro es lo contrario, una unificación.

4.3. Gestión de hilos. Espera de hilos

Ejemplos join

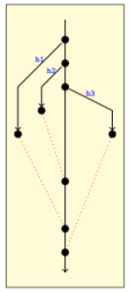
```
package ejemplosHilos;

public class EjemploJoin1 {

            public static void main (String[] args) throws InterruptedException {

                        Thread h1 = new Thread(new NuevoEjemploJoin1 ( "1" ));

                        Thread h2 = new Thread(new NuevoEjemploJoin1 ( "2" )); Thread h3 = new Thread(new
                        NuevoEjemploJoin1 ( "3" ));
                        h1. start ();
                        h2. start ();
                        h3. start ();
                        h1. join ();
                        h2. join ();
                        h3. join ();
      }
}
```



h1.start(): h1 se ejecuta.
h2.start(): h2 se ejecuta.
h3.start(): h3 se ejecuta.
h1.join(): esperamos a que h1 pare.
h2.join(): esperamos a que h2 pare.
h3.join(): esperamos a que h3 pare (no hace falta).

4.3. Gestión de hilos. Espera de hilos

Ejemplos join

```
package ejemplosHilos;
public class NuevoEjemploJoin1 implements Runnable{
        String mensaje;
                public NuevoEjemploJoin1(String nombre){
                    mensaje=nombre;
                }

                public void run(){
                        System.out.println("Hola soy el hilo "+ mensaje);
                }

}
```

4.3. Gestión de hilos. Espera de hilos

Ejemplos join

Reunión de alumnos. El siguiente ejemplo usa Threads para activar simultáneamente tres objetos de la misma clase, que comparten los recursos del procesador peleándose para escribir a la pantalla.

```
public static void main(String args[]) throws InterruptedException{
        Thread juan = new Thread (new Alumno("Juan"));
        Thread luis = new Thread (new Alumno("Luis"));
        Thread nora = new Thread (new Alumno("Nora"));

        juan.start();
        juan.join();

        pepe.start();
        pepe.join();

        jorge.start();
        jorge.join();

        }
```

Simplemente a cada instancia de Alumno(…) que creamos la hemos ligado a un Thread y puesto a andar. Corren todas en paralelo hasta que mueren de muerte natural, y también el programa principal termina.

4.3. Gestión de hilos. Espera de hilos

Ejemplos join

La clase Alumno podría ser algo así:

```
class Alumno implements Runnable{

String mensaje;

public Alumno(String nombre){ mensaje = "Hola, soy " + nombre + " y este

es mi mensaje numero: ";

} public void

run(){

      for (int i=1; i<6;i++){
                        String msj = mensaje + i;
                        System.out.println(msj);
      }

}
```

4.3. Gestión de hilos. Espera de hilos

Ejemplos join

La salida será más o menos así:

```
Hola, soy Juan y este es mi mensaje numero 1
Hola, soy Juan y este es mi mensaje numero 2
Hola, soy Juan y este es mi mensaje numero 3
Hola, soy Juan y este es mi mensaje numero 4
Hola, soy Juan y este es mi mensaje numero 5
…….etc.
```

4.3. Gestión de hilos. Espera de hilos

Ejemplos join

```
package psp_ejemplo4_thread;

import java.util.logging.Level;
import java.util.logging.Logger;

public class Ejecutador {

    public static void main(String[] args) {
        HiloThread2 ht2 = new HiloThread2();
        ht2.start(); for (int i=0;i<10;i++){
            new HiloThread(i,ht2).start();
        }
    }
}
```

4.3. Gestión de hilos. Espera de hilos

```
package psp_ejemplo4_thread;

import java.util.logging.Level; import
java.util.logging.Logger;

Public class HiloThread extends Thread { private
     int nHilo; private HiloThread2 ht2;

     public HiloThread(int _nHilo, HiloThread2 _ht2) {
          this.nHilo = _nHilo; this.ht2 = _ht2;
     } public void run() {
                                   /* Si hacemos el join de ht2 para cada hilo de HiloThread, condicionamos la ejecución de cada uno
de estos hilos a la finalización de ht2 */
//           try {
//                ht2.join();
//           } catch (InterruptedException ex) {
//                ex.printStackTrace();
//           }
          for (int i = 0; i < 100; i++) {
                          System.out.println("Hilo:" + nHilo + ". Iteracción:" + i);
          }
     }
}
```

4.3. Gestión de hilos. Espera de hilos

```
package psp_ejemplo4_thread;

public class HiloThread2 extends Thread {
    public void run(){ for (int
    i=0;i<1000;i++){
            System.out.println ("En el hilo 2:" + i);
        }
    }
}
```