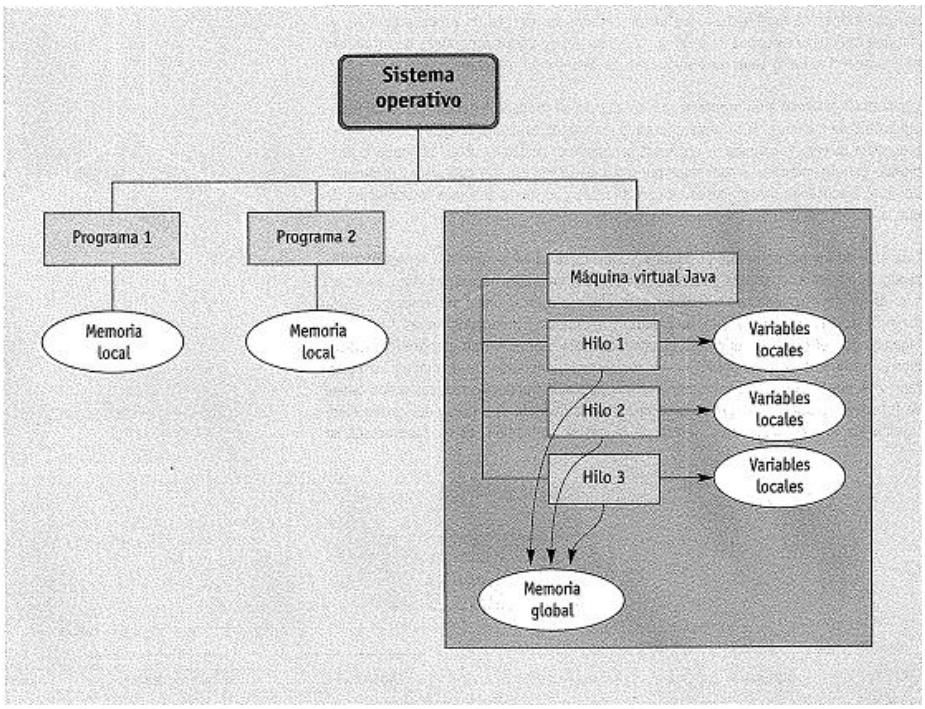
Hilos

Un hilo es una parte de un proceso, que tiene variables locales propias y comparte la memoria con el resto de hilos del mismo proceso.

La programación de aplicaciones que utilizan hilos no suele ser trivial, ya que se debe controlar que el trabajo de un hilo no interfiera con el trabajo de otro hilo en un mismo proceso. Además, en ocasiones, también es necesario que los hilos se coordinen entre ellos.

El concepto de hilo en Java es de vital importancia, por lo que el JDK incorpora herramientas y mecanismos para su creación y manipulación.



La imagen muestra cómo Java es capaz de dar la impresión de que se están ejecutando simultáneamente varias máquinas virtuales en una máquina mediante el uso de hilos. Cada hilo posee variables locales y es capaz de acceder a la memoria global compartida. La arquitectura multihilo es una característica incorporada en todos los programas Java de forma implícita. Cualquier programa Java ejecutado por la máquina virtual Java es un hilo en ejecución.

Para crear un hilo en Java existen dos formas: implementando la interfaz *Runnable* de Java o heredando de la clase *Thread* de Java.

La clase Thread

Java incorpora la clase Thread para la creación y manipulación de hilos.

El siguiente ejemplo:

Ejemplo 1

```
public class VerificarHiloImplicito {
	public static void main(String arg[]) {
		Thread hiloPrincipal = Thread.currentThread();
		System.out.println("------------------------------");
		System.out.println(hiloPrincipal.getName());
		System.out.println(hiloPrincipal.toString());
		System.out.println(hiloPrincipal .activeCount());
		System.out.println("------------------------------");
	}

}
```

Se declara un objeto de la clase *Thread* que llamamos *hiloPrincipal*. En esa misma línea se invoca el método *currentThread*, que devuelve una referencia al hilo que se está ejecutando en ese momento. El programa sigue imprimiendo por la salida estándar la información de este hilo. El resultado de la ejecución de este programa sería:

```
------------------------------
main
Thread[main,5,main]
1
------------------------------
```

De manera que cualquier programa que se ejecuta en Java es un hilo (o más de uno).

En el siguiente ejemplo se crean dos clases:

Ejemplo 2

```
public class SimpleThread1 extends Thread{
	public SimpleThread1(String nombre){
		super(nombre);
	}
	public void run(){
		for(int i=0; i<10;i++){
			System.out.println(i + " " + getName());
		}
		System.out.println("Fin! " + getName());
	}

}
public class DosThreadsDemo1 {
	public static void main (String[] args){
		SimpleThread1 hilo1 = new SimpleThread1("Hilo1");
		SimpleThread1 hilo2 = new SimpleThread1("Hilo2");
		hilo1.start();
		hilo2.start();
```

```
	}
}
```

La primera crea una clase *Hilo* a partir de la clase *Thread*, y la segunda es el programa principal, que crea dos clases *Hilo*.

La clase *SimpleThread1* hereda de la clase *Thread*, redefiniendo métodos de ésta. El más importante es el método run(). En este método se codifica la tarea del hilo. En el anterior ejemplo el hilo imprime 10 veces su nombre por la salida estándar. Al heredar de la clase *Thread* tenemos accesibles sus métodos, entre ellos el método *getName()*, que devuelve un String con el valor introducido por el constructor de la clase. El constructor de la clase *SimpleThread1* invoca con *super()* al constructor de la clase de la que hereda.

Lo métodos constructores que existen para la clase *Thread* son los que se indican a continuación.

```
package java.lang;
public class Thread implements Runnable {
	public Thread();
	public Thread(Runnable target);
	public Thread(ThreadGroup group, Runnable target);
	public Thread(String name);
	public Thread(ThreadGroup group, String name);
	public Thread(Runnable target, String name);
	public Thread(ThreadGroup group, Runnable target, String name);
	public Thread(ThreadGroup group, Runnable target, String name, long stackSize);
	public void start();
	public void run();
}
```

Elementos importantes:

String name: es el nombre del hilo utilizado cuando se imprime información del objeto Hilo. Si no introducimos ningún nombre, por defecto le asignará un valor de “Thread-N”, donde N es un valor único.

Runnable target: la interfaz *Runnable* se explicará más adelante. En esta interfaz se puede codificar el conjunto de acciones que queremos que haga un hilo.

Thread group: todo hilo pertenece a un grupo. Por defecto, se asigna el grupo del hilo que llama al constructor.

Stack size: no se utiliza mucho, pero a veces interesa determinar el tamaño de la pila del hilo donde guardar variables temporales. Si se quiere desarrollar aplicaciones para distintas plataformas no es recomendable utilizarlo.

Observando el ejemplo 2, para poder utilizar la clase *SimpleThread1*, hemos creado la clase *DosThreadsDemo1*, en la que se crean dos instancias de la clase *SimpleThread1* (hilo1 e hilo2) y se invoca para cada una de ellas el método start() para ponerlas en ejecución.

Para ejecutar un hilo hay que llamar al método start(), este método, entre otras muchas cosas, invoca al método run() del hilo.

El resultado de ejecutar la clase *DosThreadsDemo1* es:

```
0 Hilo1
1 Hilo1
2 Hilo1
3 Hilo1
4 Hilo1
5 Hilo1
6 Hilo1
7 Hilo1
8 Hilo1
9 Hilo1
Fin! Hilo1
0 Hilo2
1 Hilo2
2 Hilo2
3 Hilo2
4 Hilo2
5 Hilo2
6 Hilo2
7 Hilo2
8 Hilo2
9 Hilo2
Fin! Hilo2
```

Mediante el método start() el hilo se registra en un módulo de la máquina virtual de Java llamado planificador de hilos, el cuál decide qué hilo se debe ejecutar en cada momento de tiempo. El planificador de hilos va recibiendo peticiones de hilos para poder ejecutar. Normalmente asigna un tiempo de CPU a cada hilo que lo solicita, utilizando una política de "Primero que entra, primero al que se sirve" (*First In First Out: FIFO*). El invocar el método start() del hilo no significa que el hilo entre en ejecución de forma inmediata. Cada hilo compite con los demás para obtener tiempo de la CPU para ejecutarse. El planificador de procesos asigna los tiempo de ejecución para cada uno de los hilos y durante el ciclo de vida del hilo pasará tiempo ejecutándose y tiempo en otros estados.

En la salida del ejemplo anterior el Hilo2 ha terminado antes que el Hilo1, pero probablemente, si se ejecuta varias veces el programa, cada caso obtenga salidas distintas. Según el ordenador desde donde se ejecute el programa y la carga que tenga en ese momento, a un hilo le puede dar tiempo a terminar su trabajo (el código del método run()) o realizar parte de él antes de consumir el tiempo asignado por el planificador.

La interfaz Runnable

Otra forma de crear un hilo es mediante la interfaz *Runnable*. Esta interfaz solo posee el método *run()*.

Ejemplo 3

```
public class SimpleThreadRunnable implements Runnable{
	String nombre;
	public SimpleThreadRunnable(String nom){
		this.nombre=nom;
	}
	public void run(){
		for (int i=0;i<10;i++){
			System.out.println(i + " " + nombre);
		}
		System.out.println("Fin!  " + nombre);
	}

}
```

```
public class DosThreadsDemo {
	public static void main(String[] args){
		SimpleThreadRunnable hilo1 = new SimpleThreadRunnable("Hilo 1");
		SimpleThreadRunnable hilo2 = new SimpleThreadRunnable("Hilo 2");
		Thread h1 = new Thread(hilo1);
		Thread h2 = new Thread(hilo2);
		h1.start();
		h2.start();
	}

}
```

En este ejemplo el programa principal crea dos instancias de la clase *SimpleThreadRunnable* que implementa la interfaz *Runnable*. También define dos instanciar de la clase *Thread* y utiliza como parámetros los objetos de la interfaz *Runnable*. Lo demás es análogo al ejemplo 2.

Gestión de los hilos

Java incorpora herramientas para poder cambiar un hilo a los diferentes estados. En este apartado se explican los mecanismos para poder pasar del estado *ejecutable* a los demás estados.

Yield

Un hilo puede ofrecer voluntariamente su tiempo de CPU a otros hilos, invocando al método *yield()*, pasa así al estado "*preparado para ejecución*". Si el planificador de hilos observa que no hay ningún hilo esperando para utilizar la CPU, volverá a cambiar el estado del proceso a "*ejecutable*", concediéndole tiempo de CPU.

En el siguiente ejemplo se ha añadido el método *yield()* para comprobar cómo funciona.

Ejemplo 4

```
public class SimpleThread2 extends Thread{
	public SimpleThread2(String nombre){
		super(nombre);
	}

	public void run(){
		for(int i=0; i<10;i++){
			System.out.println(i + " " + getName());
			yield();
		}
		System.out.println("Fin!" + getName());
	}

}


public class DosThreadsDemo {
	public static void main(String[] args){
		SimpleThreadRunnable hilo1 = new SimpleThreadRunnable("Hilo 1");
		SimpleThreadRunnable hilo2 = new SimpleThreadRunnable("Hilo 2");
		Thread h1 = new Thread(hilo1);
		Thread h2 = new Thread(hilo2);
		h1.start();
		h2.start();
	}

}
```

El resultado de ejecutar el ejemplo 4 sería el siguiente

```
0 Hilo2
0 Hilo1
1 Hilo2
1 Hilo1
2 Hilo2
2 Hilo1
3 Hilo2
3 Hilo1
4 Hilo2
4 Hilo1
5 Hilo2
5 Hilo1
6 Hilo2
6 Hilo1
7 Hilo2
7 Hilo1
8 Hilo2
8 Hilo1
9 Hilo2
9 Hilo1
Fin!Hilo2
Fin!Hilo1
```

Otra ejecución muestra:

```
0 Hilo1
1 Hilo1
2 Hilo1
```

```
3 Hilo1
4 Hilo1
5 Hilo1
6 Hilo1
0 Hilo2
1 Hilo2
7 Hilo1
2 Hilo2
8 Hilo1
3 Hilo2
9 Hilo1
4 Hilo2
Fin!Hilo1
5 Hilo2
6 Hilo2
7 Hilo2
8 Hilo2
9 Hilo2
Fin!Hilo2
```

Cada vez que un hilo escribe su número y nombre da paso al otro hilo. El planificador de hilos asigna un tiempo de CPU a cada uno de los hilos. En el ejemplo 4 sólo tiene que realizar una operación de escritura del número de iteración y su nombre, y como el tiempo concedido por el planificador de hilos es suficiente para la realización de esta tarea, vemos cómo van alternándose los mensajes.

Dormido (*sleep*)

Cuando un hilo utiliza el método *sleep()* pasa al estado de *bloqueado* durante una cantidad de tiempo definida. El método *sleep()* viene definido por:

```
public static void sleep(long milisegundos) throws InterruptedException

public static void sleep(long millis, int nanos) throws InterruptedException
```

El parámetro *milisegundos* es el tiempo que permanecerá bloqueado el proceso en milisegundos. El parámetro *nanosegundos* añade la cantidad indicada en nanosegundos.

Un hilo que está *dormido* recibe un mensaje con el método *interrupt()*, pasa directamente al estado *preparado-para-ejecución* (nuevo), y cuando pasa el estado *ejecutándose*, ejecuta el código de la excepción *InterruptedException*.

Ejemplo 5

```
public class SimpleThread3 extends Thread{
	public SimpleThread3(String nombre){
		super(nombre);
	}
	public void run(){
		for(int i=0; i<10;i++){
			System.out.println(i + " " + getName());
			try {
				sleep((long)(Math.random() * 1000));
```

```
                } catch (InterruptedException e){
                }
            }
            System.out.println("Fin! " + getName());
        }

}
public class DosThreadsDemo3 {
        public static void main (String[] args){
            SimpleThread3 hilo1 = new SimpleThread3("Hilo1");
            SimpleThread3 hilo2 = new SimpleThread3("Hilo2");
            hilo1.start();
            hilo2.start();
        }

}
```

Este ejemplo crea dos hilos y los ejecuta. Los hilos escriben 10 veces el número de iteración del bucle y su nombre. En cada iteración, después de escribir su nombre, se bloquean durante un tiempo aleatorio de segundos y después vuelven a estar disponibles para su ejecución.

Con el método *Math.random()* obtenemos un valor de entre 0.0 y 0.1, que al multiplicarlo por 1000 produce una espera entre 0 y 1 segundos.

Un resultado de ejecución sería el siguiente:

```
0 Hilo1
0 Hilo2
1 Hilo1
2 Hilo1
1 Hilo2
2 Hilo2
3 Hilo1
4 Hilo1
5 Hilo1
3 Hilo2
4 Hilo2
6 Hilo1
7 Hilo1
5 Hilo2
8 Hilo1
9 Hilo1
6 Hilo2
Fin! Hilo1
7 Hilo2
8 Hilo2
9 Hilo2
Fin! Hilo2
```

Join

Un hilo puede invocar el método *join()* de otro hilo con el objeto de quedarse bloqueado a la espera de que termine el segundo hilo. Es una buena forma de crear una estructura de hilos. Puede existir un hilo padre que cree unos cuantos hilos hijo, los arranque y espere a que éstos terminen su ejecución. Un hilo bloqueado que ha invocado el método *join()* de otro hilo puede recibir un mensaje mediante la llamada a *interrupt()*, por lo que es necesario tratar la excepción *InterruptedException*.

Ejemplo 6

```
public class ThreadJoin extends Thread{
	public ThreadJoin(String nombre){
		super(nombre);
		start();
	}
	public void run(){
		try{
			for(int i=0;i<4;i++){
				System.out.println(getName() + " " + i);
				Thread.sleep(1000);
			}
			System.out.println("Fin - " + getName());
		} catch(InterruptedException e){

		}
	}

}

public class ThreadsJoinDemo {
	public static void main(String[] arg){
		System.out.println("Inicio - main");
		ThreadJoin hilo1 = new ThreadJoin("Hilo1");
		ThreadJoin hilo2 = new ThreadJoin("Hilo2");
		ThreadJoin hilo3 = new ThreadJoin("Hilo3");
		ThreadJoin hilo4 = new ThreadJoin("Hilo4");
		try{
			hilo1.join();
			hilo2.join();
			hilo3.join();
			hilo4.join();
			System.out.println("Fin - main");
		} catch(InterruptedException e){
		}
	}

}
```

Este programa crea cuatro hilos, los ejecuta y espera a que terminen. Los hilos escriben cinco veces el número de iteración del bucle y su nombre. En cada iteración después de escribir su nombre, se bloquean durante un segundo y vuelven a estar disponibles para su ejecución.

El resultado de la ejecución sería el siguiente.

```
Inicio - main
Hilo1 0
Hilo3 0
Hilo2 0
Hilo4 0
Hilo1 1
Hilo3 1
Hilo2 1
Hilo4 1
Hilo3 2
Hilo1 2
Hilo2 2
Hilo4 2
Hilo1 3
Hilo3 3
Hilo2 3
Hilo4 3
Fin - Hilo3
Fin - Hilo1
Fin - Hilo2
Fin - Hilo4
Fin - main
```