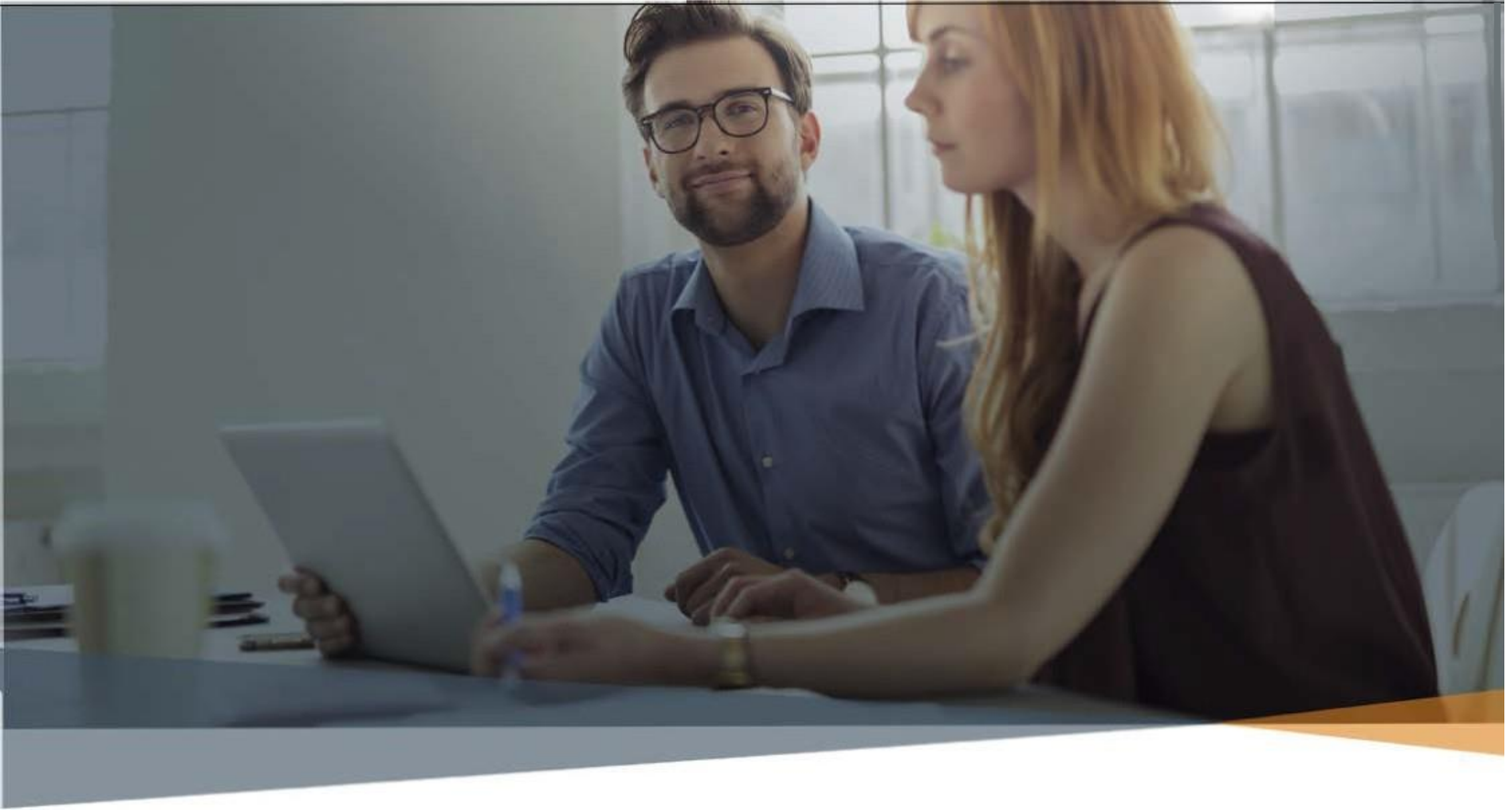
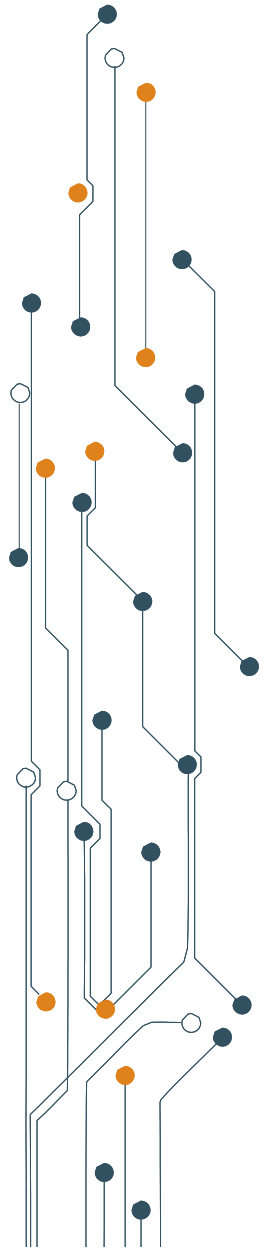


Multitarea en Java

Telefónica
EDUCACIÓN DIGITAL

Índice



1. Programación multitarea

La programación multitarea en Java consiste en organizar un programa para que realice más de una tarea a la vez, dividir el programa en subprocesos que pueden ejecutarse concurrentemente si se dispone de más de un procesador o compartiendo el tiempo del procesador.

En Java los **subprocesos** en los que se puede organizar un programa se llaman **"threads"** Un **"thread"** es un flujo de control secuencial dentro de un programa. Los programas vistos hasta este momento no son multitarea, sólo tienen un flujo de control, sólo existe un único **"thread"**.

Todos los programas tienen al menos un subproceso de ejecución, es el llamado subproceso principal, es el que se ejecuta al iniciarse el programa. Hasta este momento es el único que se ha utilizado. A partir de este subproceso principal se crean los demás.

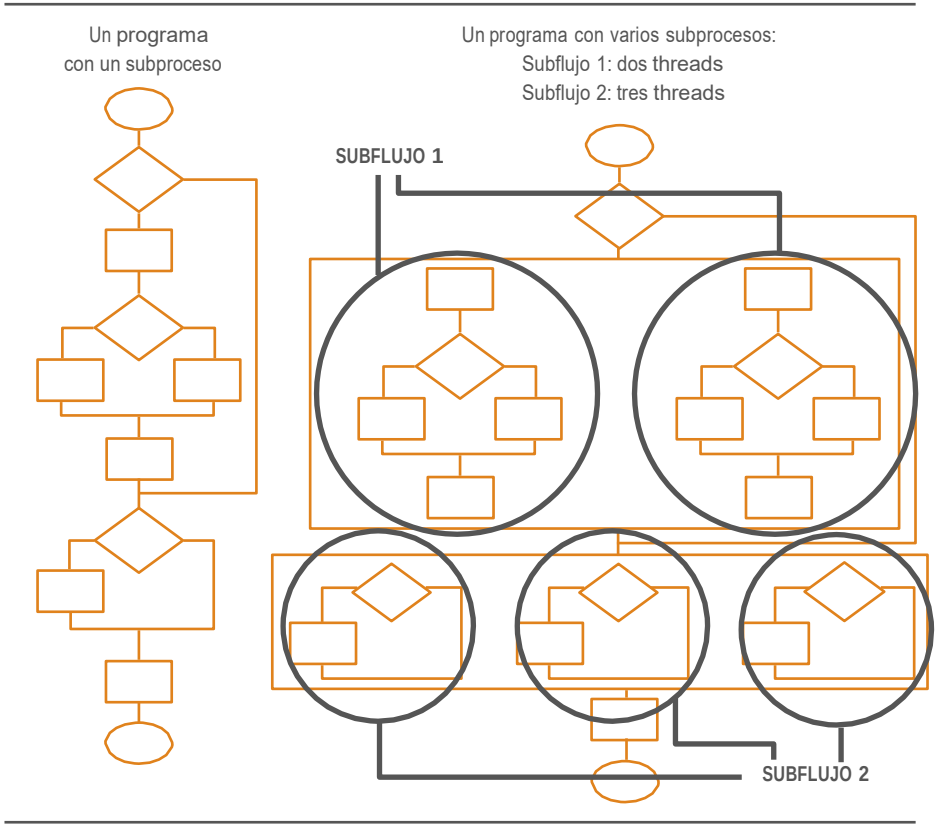


IMAGEN 12.1: MULTITAREA EN JAVA, SUBPROCESOS O THREADS.

La implementación en Java de la multitarea basada en procesos se basa en la clase **Thread** y en la interfaz ***Runnable***.

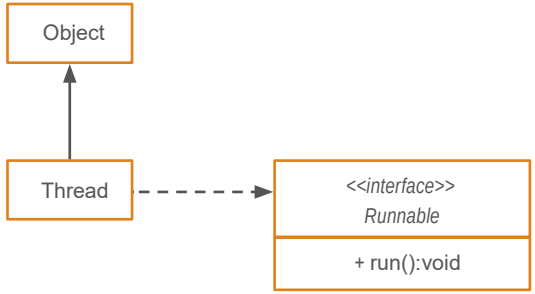


IMAGEN 12.2: CLASE THREAD E INTERFAZ RUNNABLE.

En el método **run** es donde tiene lugar la acción del subproceso, es donde se define el código que ejecutará el subproceso. Dentro del método **run** puede invocar otras funciones, usar otras clases y declarar variables al igual que el subproceso principal.

2. La clase Thread y la interfaz Runnable

Hay dos modos de proporcionar el método **run** a un subproceso:

- En una subclase de **Thread** sobrepasando el método **run**.
- En cualquier clase implementando el interface **Runnable** y definiendo su función **run**.

La razón de que existan estas dos posibilidades es porque en Java no existe la herencia múltiple. Si una clase deriva de otra no podemos hacer que derive también de **Thread**.

Creando una subclase de **Thread** o implementando ***Runnable*** sería básicamente de la forma siguiente:

```
public class Subproceso extends Thread {
//public class Subproceso implements Runnable{
    public Subproceso(String nombre) {
          super(nombre);
    }
    public void run(){
          //definir run...
    }
}
```

```
//En otro código, por ejemplo una función main
//se crean diferentes subprocesos
public static void main(String [] args){
          Subproceso subproceso1 = new Subproceso(“Subproceso 1”);
          Subproceso subproceso2 = new Subproceso(“Supproceso 2”);
          //otras sentencias
          subproceso1.start();//start inicia el proceso llamando a run
          subproceso2.start();//start inicia el proceso llamando a run
          //new Thread(subproceso1).start();
          //new Thread(subproceso2).start();
          //otras sentencias
}
```

Los principales métodos definidos en la clase **Thread** son los que se indican en la tabla siguiente:

Método	Funcionalidad
void run()	Punto de inicio del subproceso.
void start()	Inicia la ejecución de un subproceso invocando a su método run().
final void join()	Se detiene el actual y espera a que termine otro subproceso.
final void interrupt()	Interrumpe el subproceso.
final String getName()	Obtiene el nombre de un subproceso.
final setName(String nombre)	Define el nombre del subproceso.
long getId()	Devuelve el ID del subproceso, long generado cuando se creó.
final String getPriority()	Obtiene la prioridad de un subproceso.
final setPriority(int prioridad)	Cambia la prioridad del subproceso.
final boolean isAlive()	Obtiene true si un subproceso sigue en ejecución.
final boolean isInterrupted()	Obtiene true si un subproceso ha sido interrumpido.
static void sleep(long milis)	Suspende un subproceso durante los milisegundos especificados.
static Thread currentThread()	Devuelve una referencia al subproceso que se esta ejecutando.
Thread.State getState()	Devuelve el estado actual del subproceso.

Tabla 12.1: Métodos de la clase Thread.

En el ejemplo siguiente, se crean dos clases de “threads”, uno con subclase de **Thread**, otro implementando ***Runnable***. Se crean dos objetos de cada tipo y se lanza su ejecución con **run**. En cada **run** se ejecuta un código de información de que thread es un número determinado de veces.

```
public class ClaseThread extends Thread {
    private int n;
    public ClaseThread(int n) {
          this.n = n;
    }
          public void run() {
          int i = 1;
          while (true) {
                System.out.println(“Run del thread “ +
this.getName()
                + “ de “ + getClass().getSimpleName()
+ “ con ID “
                + this.getId()    + “. Iteracion “ +
i++);
                if (i > n) {
                      break;
                }
          }
    }
```

```
public class ClaseRunnable implements Runnable{
    private int n;
    public ClaseRunnable(int n){
          this.n=n;
    }
    @Override
    public void run() {
        int i = 1;
        while (true) {
            System.out.println(“Run del thread “+
            getClass().getSimpleName()
                        +”. Iteracion “ + i++);
            if (i > n) {
                break;
            }
        }
    }
public static void main(String[] args) {
          Random aleatorio = new Random();
        ClaseThread t1= new ClaseThread(aleatorio.
nextInt(5)+1);
        ClaseThread t2= new ClaseThread(aleatorio.
nextInt(6)+1);
        ClaseRunnable r1=
                                        new
ClaseRunnable(aleatorio.nextInt(4)+1);
        ClaseRunnable r2=
                                        new
ClaseRunnable(aleatorio.nextInt(7)+1);
        t1.start();
        t2.start();
        new Thread(r1).start();
        new Thread(r2).start();
    }
```

3. Subprocesos. Ciclo de vida

Un subproceso desde que se crea como objeto "**runnable**", hasta que se libera el espacio que ocupa en memoria, pasa por una serie de estados. Los estados de un subproceso son los que están indicados en la enumeración **Thread.State**. La función **static getState()** de **Thread** devuelve el estado actual del subproceso en ejecución.

Los estados de un proceso son:

Estado	Funcionalidad
NEW	Se ha creado el subproceso pero todavía no ha arrancado.
RUNNABLE	El subproceso se está ejecutando en la JVM. De acuerdo con el planificador de tareas (scheduler) del sistema operativo está en ejecución o preparado para seguir ejecutándose, de acuerdo a su prioridad, ocupando un "slice" de tiempo del procesador. El planificador de tareas tiene que asegurar que todos los subprocesos en este estado tienen que llegar a ejecutarse.
BLOCKED	El subproceso se encuentra bloqueado, esperando a volver a ejecución.
WAITING	El subproceso se encuentra esperando indefinidamente, a que otro realice una determinada acción que lo saque de este estado.
TIMED_ WAITING	El subproceso se encuentra esperando durante un tiempo determinado.
TERMINATED	El subproceso ha finalizado su ejecución, ha terminado la función run.

Tabla 12.2: Estado de un subproceso.

De acuerdo con estos estados, la imagen siguiente muestra cual es ciclo de vida de un subproceso.

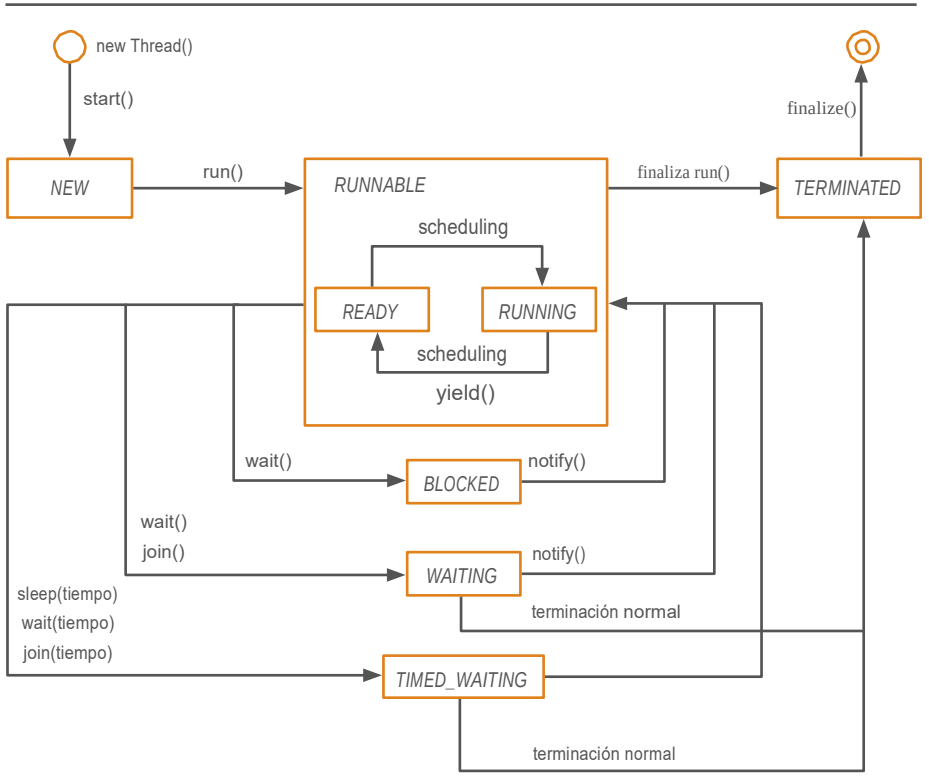
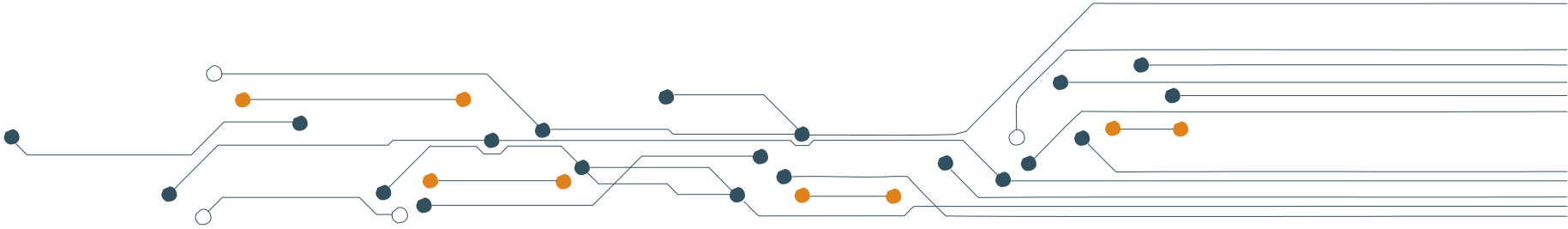


IMAGEN 12.2: CICLO DE VIDA DE UN SUBPROCESO.

Los subprocesos en Java después de ser creados pasan al estado "**NEW**" y cuando se les invoca la función **start** pasan a "**RUNNABLE**". En este estado, es cuando se ejecuta el método run, pero en programación multitarea y concurrente, el tiempo del procesador se reparte entre los subprocesos en este estado, de tal manera que se garantice que todo subproceso se ejecuta.

El planificador de tareas ("**scheduler**") es quien cuando a los subprocesos les llega su turno, por la prioridad que tengan establecida, los pasa a "**RUNNING**", estando en este estado el tiempo de procesador correspondiente ("**slice**") y cuando acabe dicho tempo pasan a "**READY**". Así seguirá cada subproceso hasta que termine la ejecución de **run**, pasando a "**TERMINATED**" o se produzca algunas de las situaciones siguientes:

- En método **run**, se ejecuta la función **Thread.sleep**, el subproceso cambia a "**TIMED_WAITING**". Permanece en este hasta que terminan los milisegundos pasados como parámetro a esta función.
- Cuando desde otro subproceso se invoca a las funciones wait, el subproceso actual, pasa al estado "**WAITING**". Permanece en este hasta que desde otro subproceso se invoque la función **notify**. Tanto **wait** como **notify** son funciones heredadas de **Object**.
- Cuando desde otro subproceso se invoca a la función **Thread.join**, el subproceso actual, pasa al estado "**WAITING**". Permanece en este hasta que el otro subproceso termina.
- Cuando desde otro subproceso en un bloque sincronizado, se invoca a la función **wait** , el subproceso actual, pasa al estado "**BLOCKED**". Permanece en este hasta que desde otro subproceso se invoca la función **notify**.
- Si el subproceso actual recibe la función **yield**, si esta en "RUNNING", pasa a "READY". esto significa que deja la ejecución y pasa a la cola de subprocesos en espera de volver a ejecución.
- Si desde un subproceso se invoca a la función **interrupt** del subproceso actual y este esta en "**WAITING**" o "**TIME_WAITING**", se produce la interrupción **InterruptException**, se procesa el bloque **catch** que pudiera tener asociada, y a continuación sigue con la ejecución de **run** que quedara.



En el ejemplo se siguiente se crean dos subprocesos y se cambia su estado con función **sleep** y se provoca su parada con **interrupt**.

```
public static void main(String[] args)
                    throws InterruptedException {
        Prueba p1 = new Prueba("---thread 1---");
        Prueba p2 = new Prueba("---thread 2---");
        System.out.println("ESTADO DE "+p1.
getName()+
                                " ===> "+
p1.getState());
        System.out.println("COMIENZO: " + System.
currentTimeMillis());
        p1.start();
        System.out.println("ESTADO DE "+p1.
getName()+
                            " ===> "+ p1.getState());
        p2.start();
        Thread.sleep(100);
        p1.interrupt();
        Thread.sleep(100);
        p1.interrupt();
        p2.interrupt();
        Thread.sleep(100);
        p2.interrupt();
        int i=0;
        do{    } while(!(    p1.getState() ==
Thread.State.TERMINATED &&
                    p2.getState() == Thread.State.
TERMINATED));
        System.out.println("ESTADO DE "+p1.
getName()+
                            " ===> "+ p1.getState());
        System.out.println("FINALIZACION: " +
                    System.currentTimeMillis());
    }
```

La función run de estos subprocesos es la siguiente:

```
public void run() {
    for (int i = 1; i <= 3; i++) {
          System.out.println(“INICIO DE CICLO “+ i +
                “ DE “ + getName() + “ ===> “+
                (System.currentTimeMillis() - inicio)
                            + “ milisegundos”);
          try {
                Thread.sleep(1000);
          }
          catch (InterruptedException e) {
                System.out.println(“INTERRUMPIDO: “ +
getName()
                + “ ===> “ + (System.
currentTimeMillis() - inicio)
                            + “ milisegundos”);
          }
          finally {
                      System.out.println(“FIN DE CICLO
“+ i +
                      “ DE “ + getName() + “ ===> “

                            (System.
currentTimeMillis() - inicio)
                                  + “ milisegundos”);
                }
          }
    }
```

Las funciones **run** y **main** son miembro de la misma clase que hereda de **Thread**:

```
package com.juan.threads02.interrup;
public class Prueba extends Thread {
    private long inicio;
    public Prueba(String str) {
          super(str);
          inicio = System.currentTimeMillis();
    }
    public void run() { /*sentencias*/ }
    public static void main(String[] args)
          throws InterruptedException { /*sentencias*/
}
}
```

4. Finalización de subprocesos

Para saber si un subproceso ha finalizado se puede utilizar la función **isAlive()**, devuelve true si aún no ha finalizado y sigue en ejecución. En el código siguiente se utiliza para que el proceso principal (función main) finalice en cuanto finalicen los tres subprocesos que se arrancan en este.

```
public static void main(String args[]) {
         System.out.println("Comienza función main");
         MiThread mt1 = new MiThread("Subproceso
#1");
         MiThread mt2 = new MiThread("Subproceso
#2");
         MiThread mt3 = new MiThread("Subproceso
#3");
        do {
           System.out.print(".");
           try {
             Thread.sleep(100);
           }
           catch(InterruptedException exc) {
             System.out.println("Interrumpido en
main.");
           }
         } while (mt1.thread.isAlive() ||
                  mt2.thread.isAlive() ||
                  mt3.thread.isAlive());
         System.out.println("Finaliza la función
main");
    }
```

También se puede esperar que finalice un subproceso utilizando la función **join()**. Esta función hace que el subproceso que invoque a **join** de otro subproceso se quede esperando hasta que este finalice. Por ejemplo, si en la función **main** se invoca a **join** de otro subproceso, el código de la función **main** se queda esperando hasta que finalice el subproceso para el que se invoca **join**. En realidad, es unir a la vida de un subproceso a la de otro u otros, de ahí su nombre.

Sus formas son:

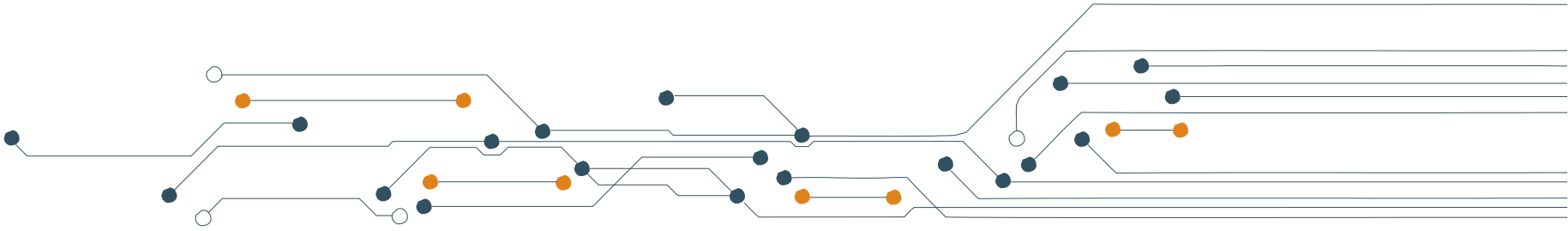
```
public final void join() throws InterruptedException
```

También se puede usar con estas dos formas, en las que se espera a que finalice o que pase el tiempo establecido en los parámetros.

```
public final void join(long millis) throws
InterruptedException
public final void join(long millis, int nanos) throws
InterruptedException
```

En el ejemplo el subproceso de main espera a que finalicen tres subprocesos a los que invoca join.

```
try {
  mt1.thread.join();
  System.out.println("Subproceso #1 se ejecuta join.");
  mt2.thread.join();
  System.out.println("Subproceso #2 se ejecuta join.");
  mt3.thread.join();
  System.out.println("Subproceso #3 se ejecuta join.");
} catch (InterruptedException exc) {
  System.out.println("Interrumpido main");
}
System.out.println("Finaliza la función main");
```



5. Sincronización

Cuando varios subprocesos están accediendo al mismo recurso puede ocurrir lo que se denomina **lecturas erróneas**. Por ejemplo, si un subproceso debe actualizar el valor de una variable, si ese cambio todavía no se ha producido y otro subproceso ha leído dicho valor, el resultado es que el segundo subproceso está trabajando con un dato erróneo, lo correcto hubiese sido esperar a que el primer subproceso actualizase el valor, para que el segundo lo leyera.

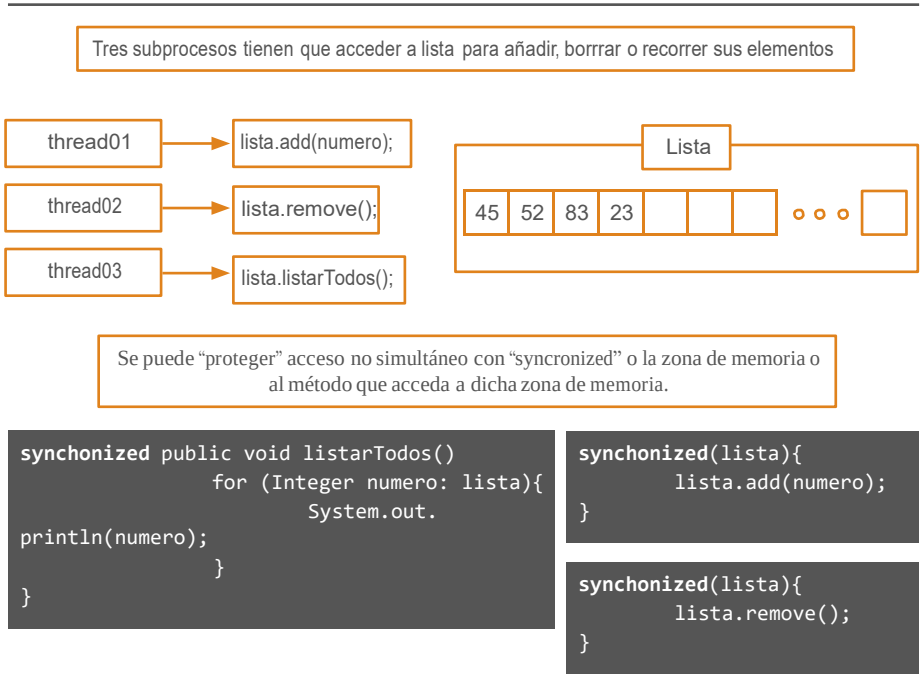


IMAGEN 12.3: SINCRONIZACIÓN CON SYNCHRONIZED.

Para evitar esto se utiliza la sincronización, implementada en Java con la palabra reservada **synchronized**, que permite sincronizar una función o un bloque de código. Así si un primer subproceso accede al método o código sincronizado bloquea a los demás subprocesos. Cuando el subproceso finaliza su ejecución desbloquea a los demás para que uno pueda acceder.

En el código siguiente, el método **sumarArray** esta sincronizado. Dos subprocesos en su función run invocan a este método, por tanto sólo uno está ejecutando el código de **sumarArray**, el otro espera a que el anterior acabe.

```
class SumarArray {
    private int suma;
    synchronized int sumarArray(int nums[]) {
          suma = 0;
          for (int i = 0; i < nums.length; i++) {
               suma += nums[i];
              System.out.println("La suma en el for
para "
                        + Thread.currentThread().
getName() + " es " + suma);
                try {
                      Thread.sleep(10);
                } catch (InterruptedException exc) {
                      System.out.println(exc.
getMessage());
                }
          }
          return suma;
    }
}
class MiThread implements Runnable {
    Thread thread;
    static SumarArray sa = new SumarArray();
    int numeros[]; int suma;
    MiThread(String nombre, int numeros[]) {
          thread = new Thread(this, nombre);
          this.numeros = numeros;
          thread.start();
    }
    public void run() {

          int suma;
          System.out.println(thread.getName() + "
EMPEZANDO");
          suma = sa.sumarArray(numeros);
          System.out.println("La suma en " + thread.
getName()
                              + " es " + suma);
          System.out.println(thread.getName() + "
TERMINANDO");
    }
}
```

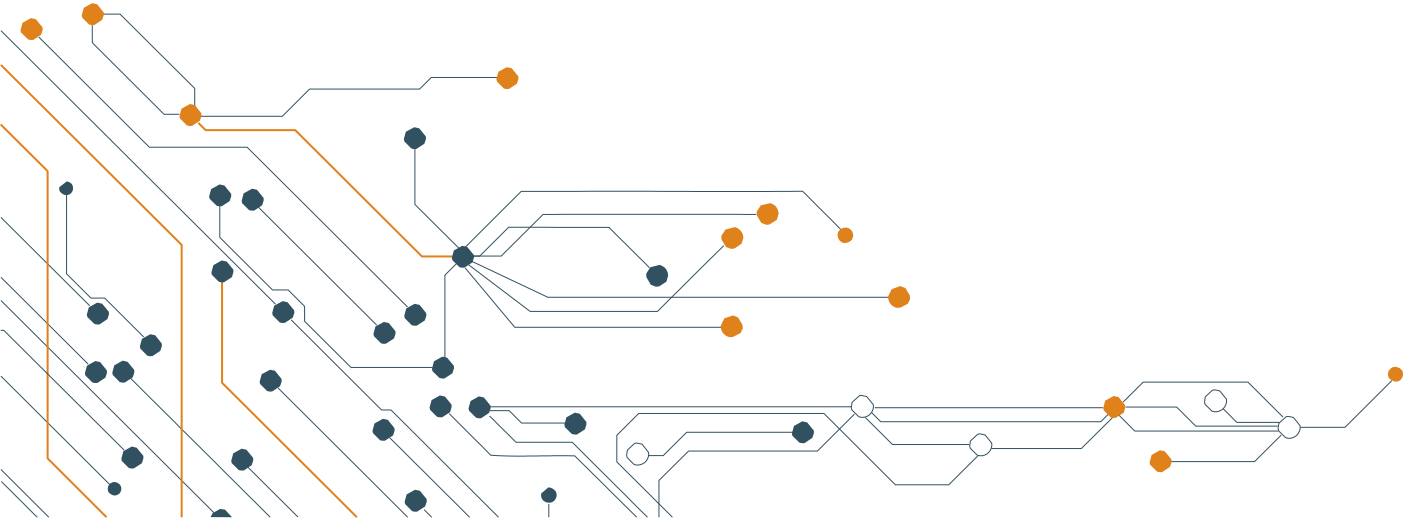
En vez de sincronizar todo un método como en el ejemplo anterior, se pueden sincronizar el bloque de código en el que se invoca al método. Esto tiene que ser así obligatoriamente, si el método a utilizar es de una clase de una librería de terceros.

El formato de esta utilización es:

```
synchronized(<referencia_objeto_a_bloquear>{
  //sentencias que invocan a metodo que accede
  //a objeto declarado entre los paréntesis
}
```

En el ejemplo anterior, se quitaría **synchronized** de la definición del método sumarArray y se añadiría el bloque sincronizado para el acceso al array de números en la llamada a la función.

```
public void run() {
    int suma;
    System.out.println(thread.getName() + “
EMPEZANDO”);
    synchronized(numeros){
          suma = sa.sumarArray(numeros);
    }
    System.out.println(“La suma en “ +
          thread.getName() + “ es “ + suma);
    System.out.println(thread.getName() + “
TERMINANDO”);
}
```



6. Comunicación de subprocesos: notify, notifyAll y wait

Cuando un subproceso intenta acceder a un recurso y no puede conseguirlo, puede ponerse a la espera, hasta que desde otro subproceso se le notifique que dicho recurso esta liberado. Por ejemplo, un subproceso intenta añadir un elemento a una colección y está llena, puede este subproceso quedarse en "WAITING" hasta que desde otro proceso le notifiquen que ya puede añadir el elemento.

Esta comunicación entre subprocesos se realiza con las funciones de la clase Object:

- **wait()**, pone en "WAITING" al subproceso que invoca la función, hasta que otro invoque a **notify** o **notifyAll**. Sus formas son:

  public final void wait() throws InterruptedException

  También se puede usar con estas dos formas, en las que se espera a que otro invoque **notify** o **notifyAll** o pase el tiempo especificado.

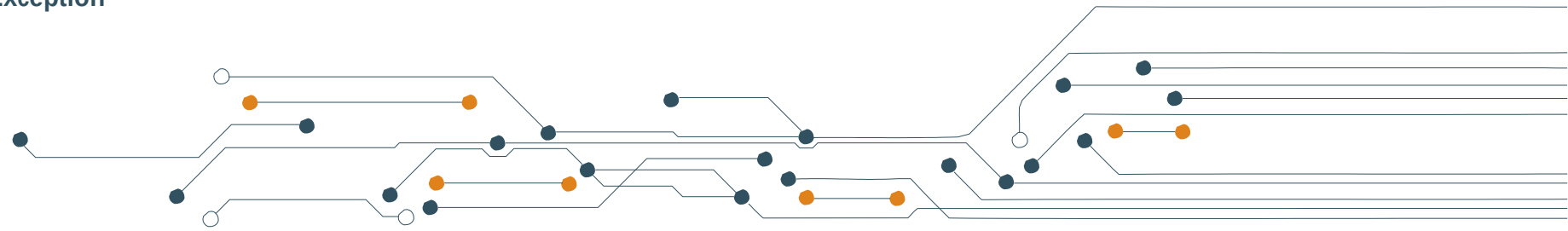
  public final void wait(long timeout) throws InterruptedException

  public final void wait(long timeout, int nanos) throws InterruptedException

- **notify()**, saca de "**WAITING**" a uno de los subprocesos que estén en dicho estado y lo pasa a "**RUNNABLE**". Su forma es:

```
public final void notify()
```

- **notifyAll()**, saca de "**WAITING**" a todos los subprocesos que estuvieran en este estado y los pasa "**RUNNABLE**"



7. Suspender, reanudar y detener subprocesos

Antes de Java 2 para suspender, reanudar y detener subprocesos se utilizaban las funciones de la clase Thread:

- final void suspend()
- final void resume()
- final void stop()

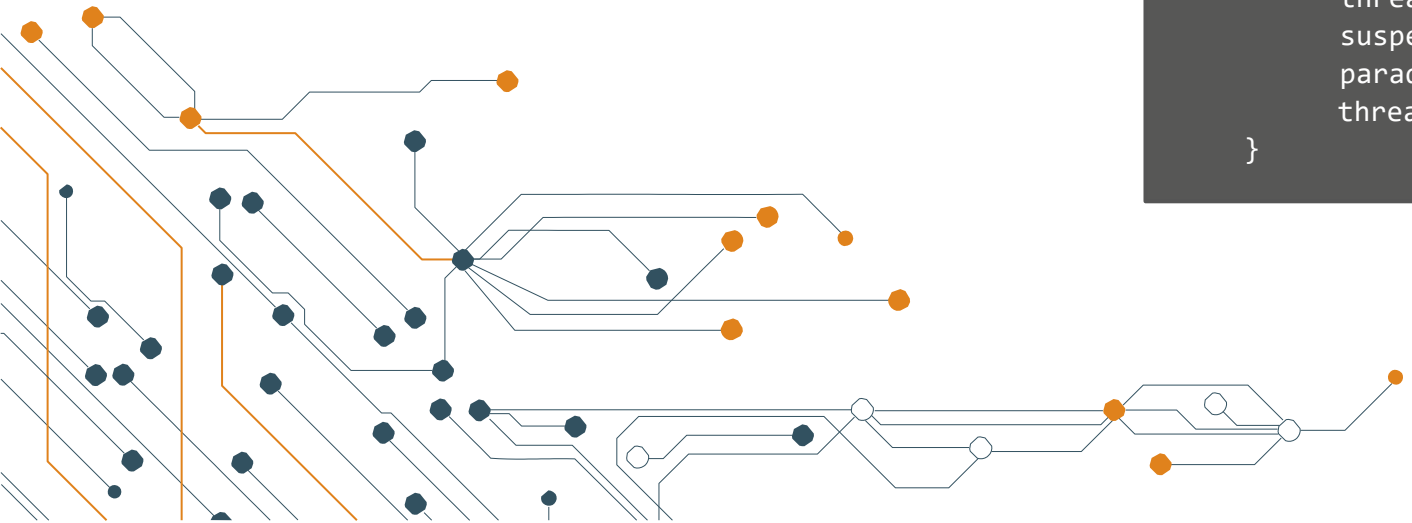
El método **suspend** y el método **stop** causaban problemas, no funcionaba adecuadamente, y se decidió ponerlos "**decrepated**", el método resume como complementario a **suspend** pasó también a este estado.

Sin utilizar estas funciones, estas operaciones se pueden realizar con comprobaciones periódicas en el método **run** sobre si tiene que suspender, reanudar o parar su propia ejecución. Se suelen utilizar dos variables, una para suspender y reanudar, la otra para parar.

En el ejemplo siguiente se muestra una posible forma de realizarlo.

```
class MiThread implements Runnable {
    Thread thread;
    boolean suspendido;
    boolean parado;

    MiThread(String nombre) {
          thread = new Thread(this, nombre);
          suspendido = false;
          parado = false;
          thread.start();
    }
```



```
public void run() {
         System.out.println(thread.getName() + “
ARRANCANDO”);
         try {
               for (int i = 1; i < 1000; i++) {
                     System.out.print(i + “ “);
                     if ((i % 10) == 0) {
                           System.out.println();
                           Thread.sleep(400);
                     }
//Bucle en run que hace las comprobaciones periódicas
//Sincronizar para acceder a suspendido y parado
                     synchronized (this) {
                           while (suspendido) {
                                 wait();
                           }
                           if (parado)
                                 break;
                     }
               }
         } catch (InterruptedException exc) {
                System.out.println(thread.getName()
+ “ INTERRUMPIDO”);
          }
         System.out.println(thread.getName() + “
FINALIZADO”);
    }
```

Las funciones en el subproceso que actualizan los valores de las variables suspendido y parado para que este pueda ser suspendido, reanudado o parado, son las que se muestran a continuación.

```
synchronized void stop() {
         parado = true;
         suspendido = false;
         notify();
    }

    synchronized void suspend() {
         suspendido = true;
    }

    synchronized void resume() {
         suspendido = false;
         notify();
    }
}
```

La funcionalidad **suspender** lleva al estado “**WAITING**”, la funcionalidad **reanudar** lleva del estado “**WAITING**” al estado “**RUNNABLE**” y la funcionalidad **parar** lleva al estado “**TERMINATED**”.

Telefónica

EDUCACIÓN DIGITAL