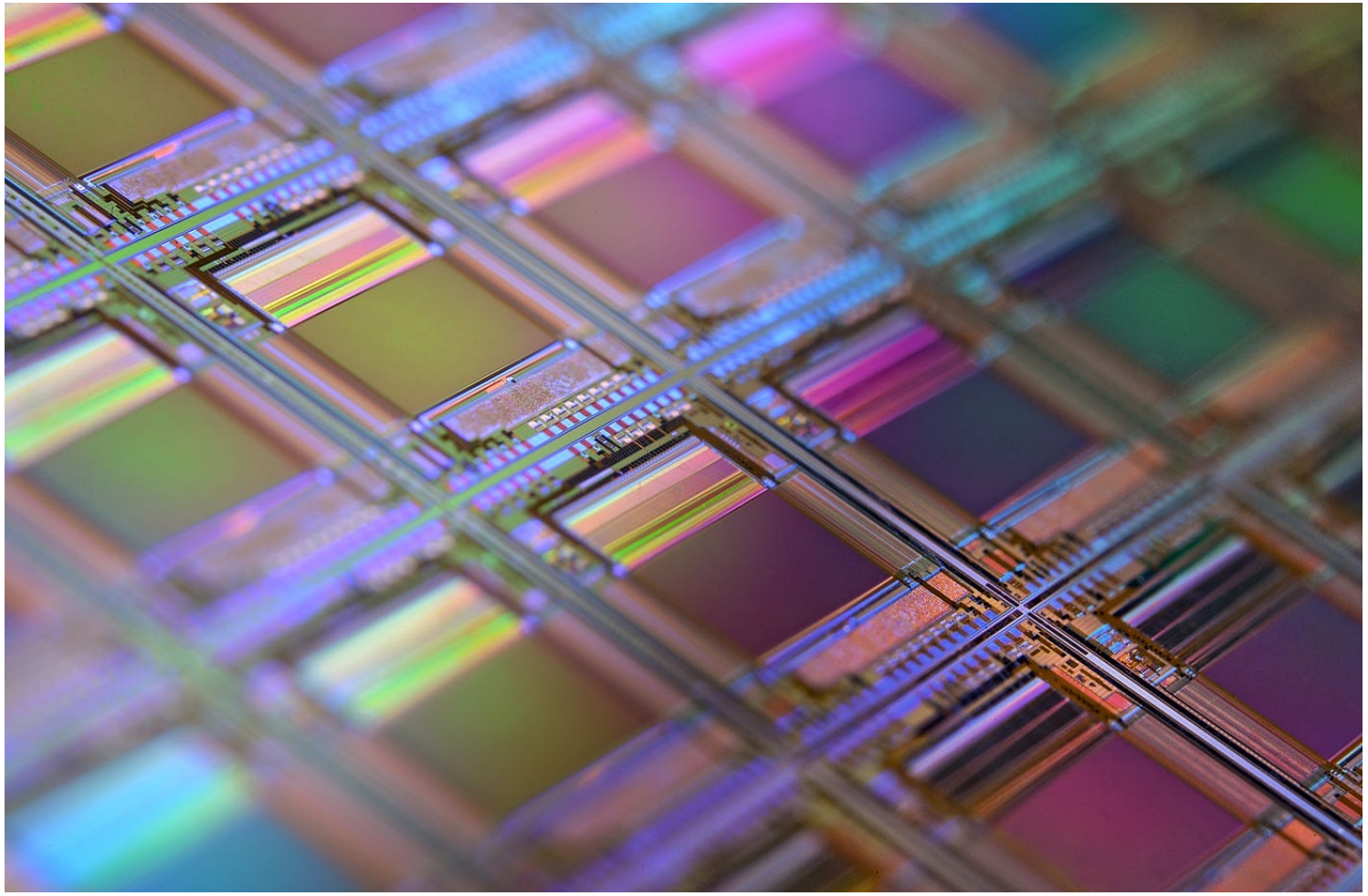
Programación de Servicios y Procesos

UT2 – Programación multihilo



PSP UT2

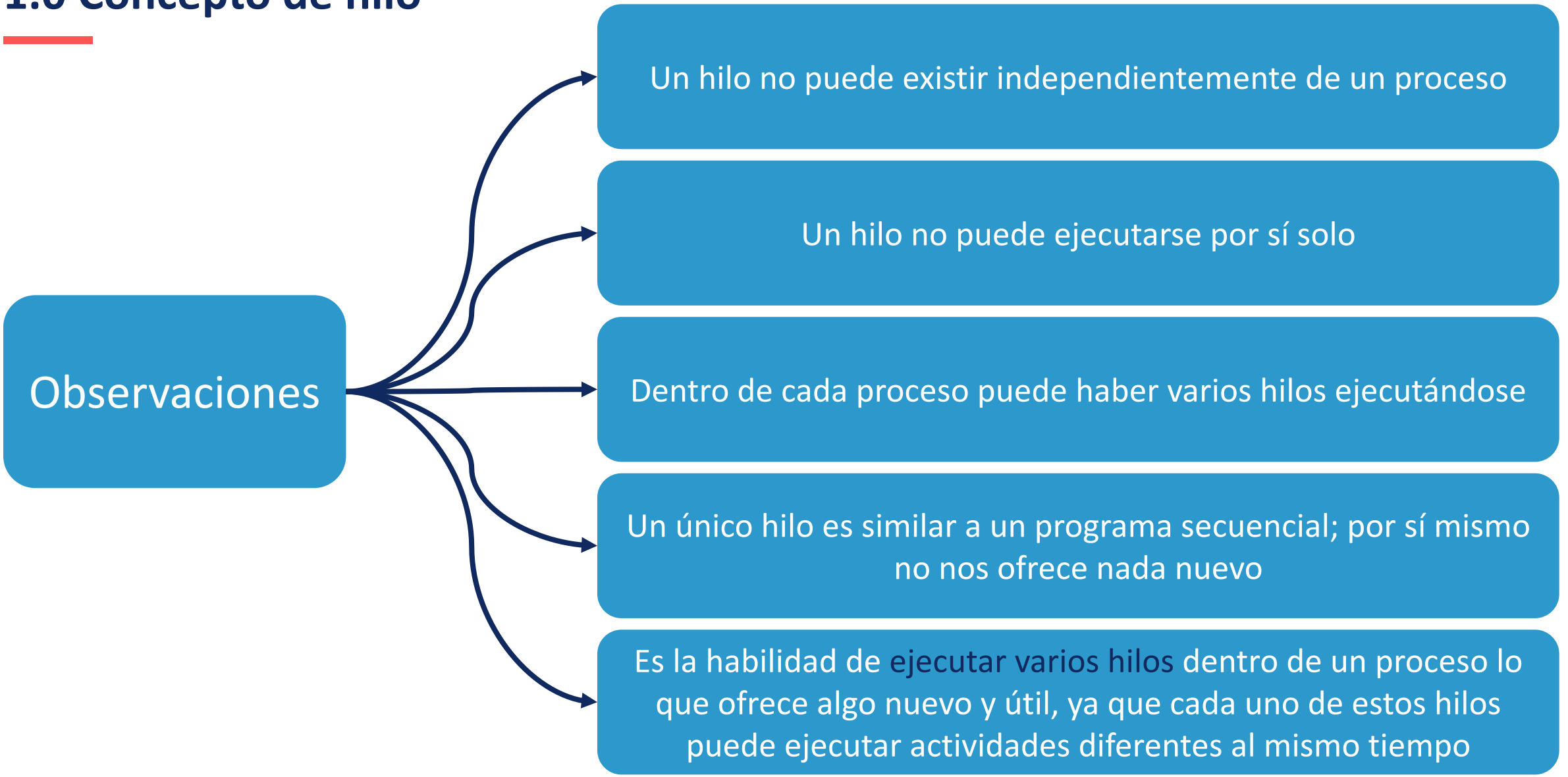
01 Progamación multihilo

CONCEPTO DE HILO

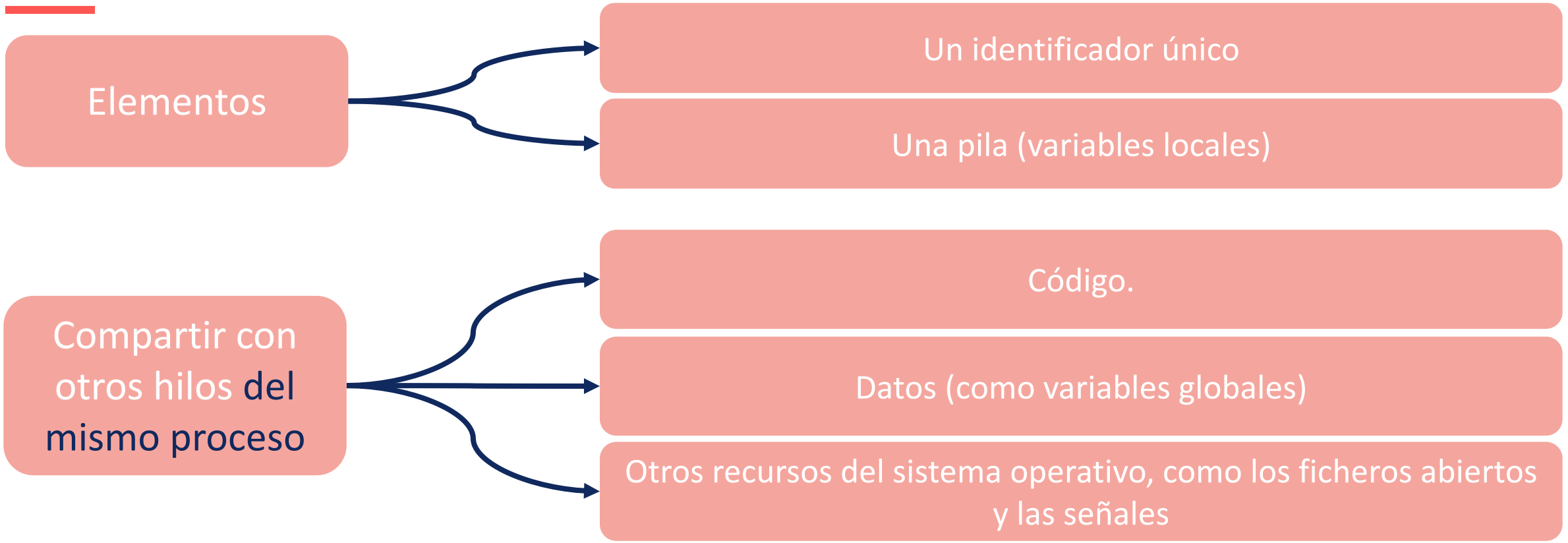
1.0 Concepto de hilo

Un hilo, denominado también subproceso, es un flujo de control secuencial independiente dentro de un proceso y está asociado con una secuencia de instrucciones, un conjunto de registros y una pila

1.0 Concepto de hilo



1.1 Recursos compartidos por los hilos



El hecho de que los hilos compartan recursos (por ejemplo, pudiendo acceder a las mismas variables) implica que sea necesario utilizar esquemas de bloqueo y sincronización, lo que puede hacer más difícil el desarrollo de los programas y así como su depuración.

1.2 Ventajas y usos de los hilos

Como consecuencia de compartir el espacio de memoria, los hilos aportan las siguientes ventajas sobre los procesos:

- Se consumen menos recursos en el lanzamiento y la ejecución de un hilo que en el lanzamiento y ejecución de un proceso
- Se tarda menos tiempo en crear y terminar un hilo que un proceso
- La conmutación entre hilos del mismo proceso es bastante más rápida que entre procesos

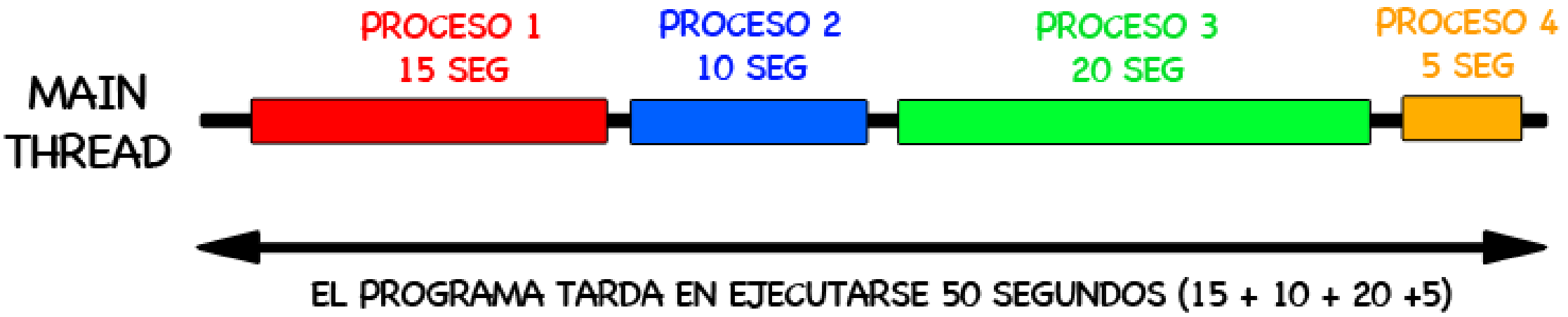
Por estas razones a los hilos se les denomina también procesos ligeros

1.2 Ventajas y usos de los hilos

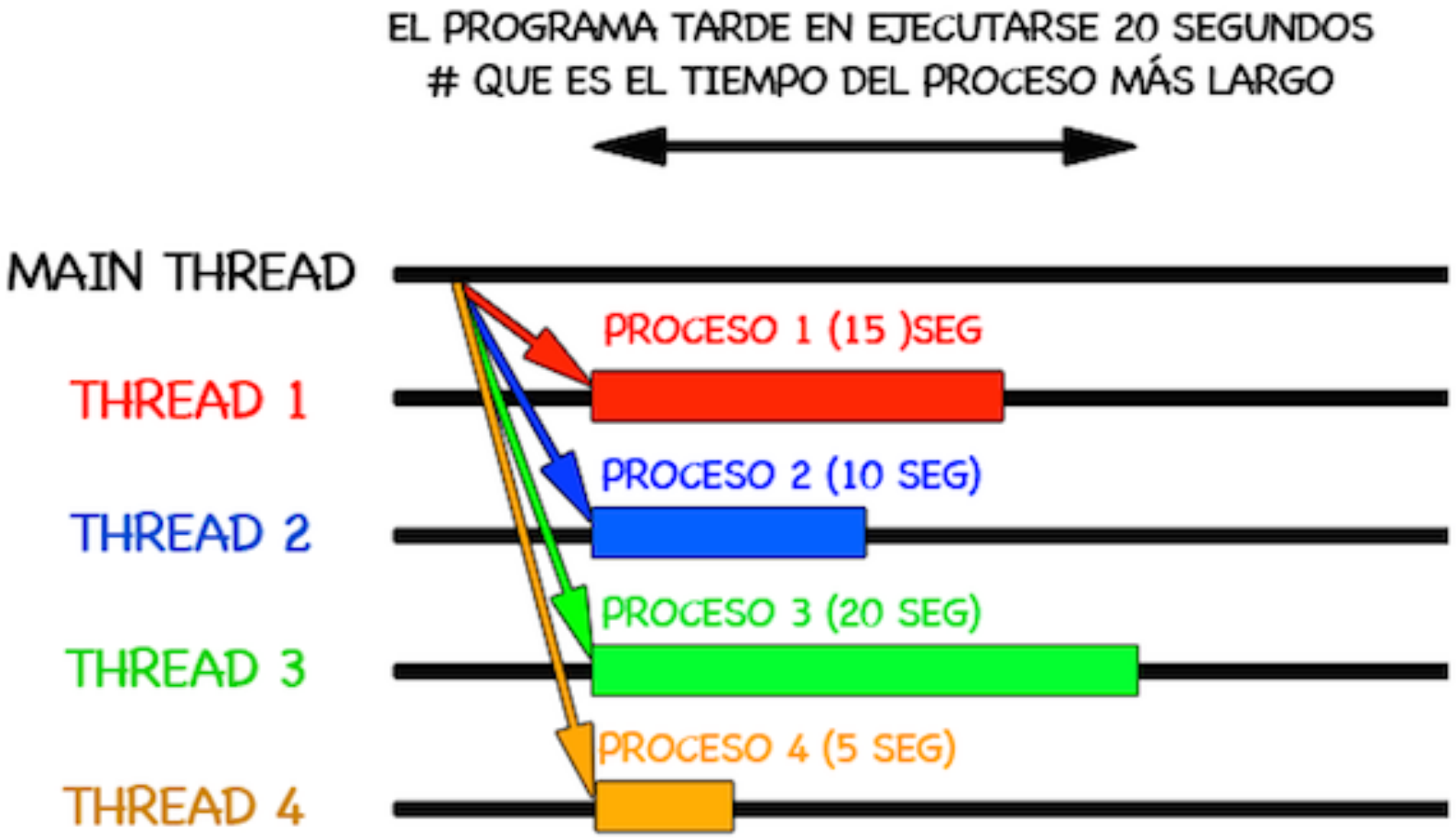
1.2 Ventajas y usos de los hilos

¿Qué es más rápido en un programa, utilizar 4 subprocesos o 4 hilos?

1.2 Ventajas y usos de los hilos



1.2 Ventajas y usos de los hilos



HILO vs PROCESO

2.0 Hilo vs Proceso

Un proceso es, en esencia, un programa que se está ejecutando.

Por lo tanto, la multitarea basada en procesos es la característica que le permite a su computadora ejecutar dos o más programas al mismo tiempo.

Por ejemplo, es una multitarea basada en procesos ejecutar el compilador de Java al mismo tiempo que utiliza un editor de texto o navega por Internet.

En la multitarea basada en procesos, un programa es la unidad de código más pequeña que puede enviar el planificador del sistema operativo

2.0 Hilo vs Proceso

En un entorno multitarea basado en hilos, el hilo es la unidad más pequeña de código distribuible.

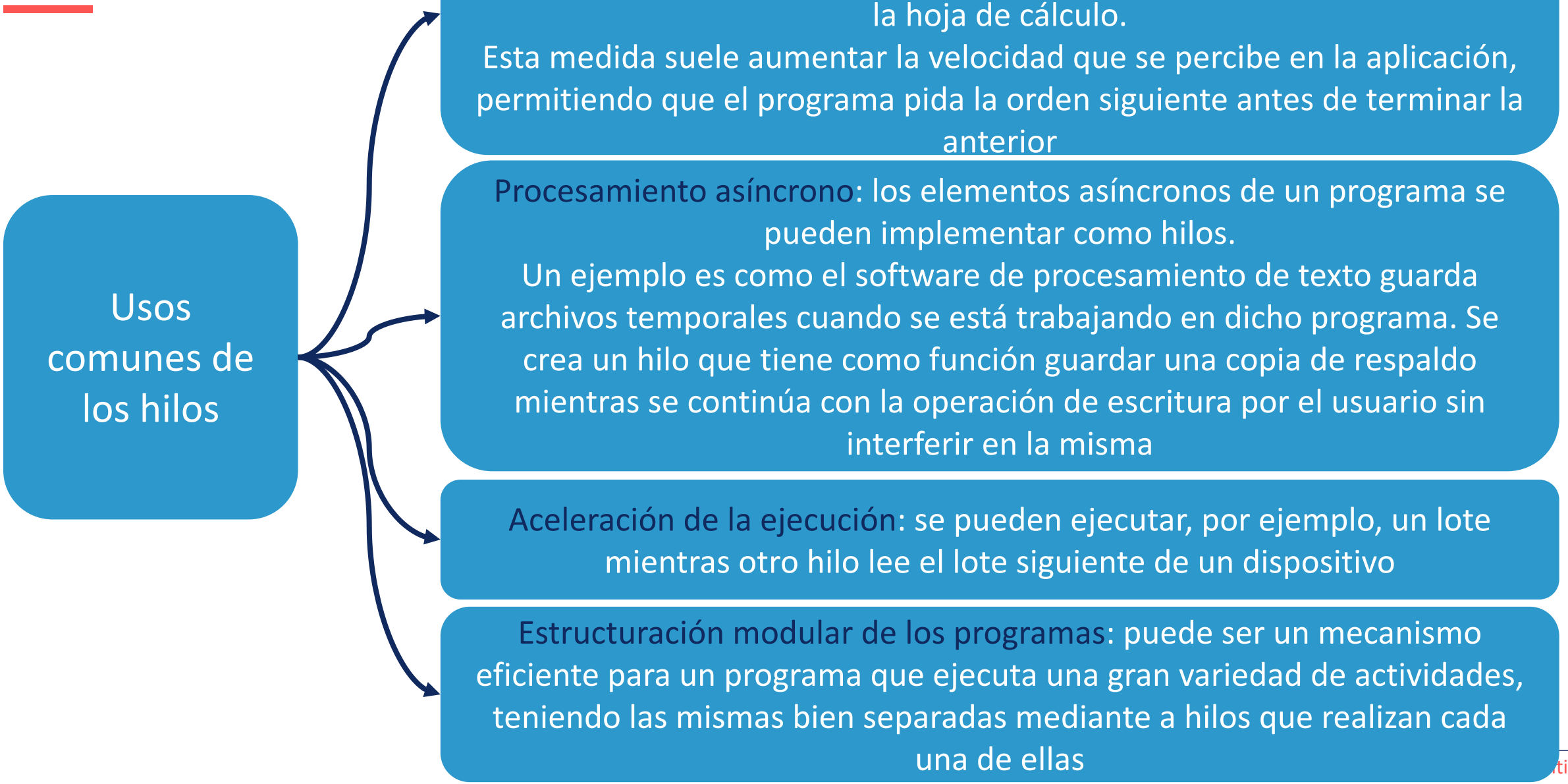
Esto significa que un solo programa puede realizar dos o más tareas a la vez.

Por ejemplo, un editor de texto puede formatear texto al mismo tiempo que está imprimiendo, siempre que estas dos acciones se realicen mediante dos hilos separados.

Aunque los programas Java utilizan entornos multitarea basados en procesos como también hemos visto en el anterior tema, la multitarea basada en procesos no está bajo el control de Java, está bajo el control del SO, mientras que la multitarea multihilo sí está bajo el control de Java.

USOS COMUNES DE LOS HILOS

3.0 Usos Comunes



CREACIÓN Y GESTIÓN DE HILOS

4.0 Creación y gestión de hilos

En java existen dos formas de crear hilos

- Extendiendo la clase ***Thread***
- Implementado la interfaz ***Runnable***

Ambas son parte del paquete ***java.lang***.

LA CLASE *THREAD*

4.1 La clase *Thread*

La forma más sencilla de añadir funcionalidad de hilo a una clase es extender la clase ***Thread***.

Esta subclase debe sobrescribir el método ***run()*** con las acciones que el hilo debe desarrollar.

La clase ***Thread*** también define otros métodos como ***start()*** y ***stop()*** (este último en desuso y desaconsejado) para iniciar y parar la ejecución del hilo.

4.1 La clase *Thread*

¿Cómo funciona?

Cuando extiendes la clase ***Thread***, debes sobrescribir el método ***run()***, que contiene el código que se ejecutará en el nuevo hilo.

Luego, puedes crear una instancia de tu clase y llamar al método ***start()*** para iniciar el hilo.

El método ***start()*** invoca el método ***run()*** en un nuevo hilo.

4.1 La clase *Thread*

```java
//Clase principal
public class MiPrimerHilo {
 public static void main(String[] args) {
      // Crear dos hilos
      MiHilo hilo1 = new MiHilo("Hilo 1");
      MiHilo hilo2 = new MiHilo("Hilo 2");

      // Iniciar los hilos
      hilo1.start();
      hilo2.start();
 }
}
```

4.1 La clase *Thread*

```java
package Ejemplos;

//Extensión de la clase Thread
class MiHilo extends Thread {
 private String nombre;

 // Constructor
 public MiHilo(String nombre) {
     this.nombre = nombre;
 }

 // Sobrescribir el método run
 @Override
 public void run() {
     for (int i = 0; i < 5; i++) {
         System.out.println(nombre + " - Contador: " + i);
         try {
             Thread.sleep(1000); // Pausa el hilo por 1 segundo
         } catch (InterruptedException e) {
             System.out.println("Hilo interrumpido: " + e.getMessage());
         }
     }
 }
}
```

4.1 La clase *Thread*

```java
//Extensión de la clase Thread
class MiHilo extends Thread {
 private String nombre;

 // Constructor
 public MiHilo(String nombre) {
     this.nombre = nombre;
 }
```

Definición de la clase ***MiHilo***:

- Esta clase extiende ***Thread***.
- Tiene un constructor que toma un nombre, que en este caso se lo hemos pasado como parámetro, para identificar el hilo

4.1 La clase *Thread*

```java
// Sobrescribir el método run
@Override
public void run() {
    for (int i = 0; i < 5; i++) {
        System.out.println(nombre + " - Contador: " + i);
        try {
            Thread.sleep(1000); // Pausa el hilo por 1 segundo
        } catch (InterruptedException e) {
            System.out.println("Hilo interrumpido: " + e.getMessage());
        }
    }
}
```

Método ***run()***:

- Este método es donde se define el trabajo que realizará el hilo.
- En este caso, imprime el nombre del hilo y un contador, pausando un segundo entre cada impresión

4.1 La clase *Thread*

```java
//Clase principal
public class MiPrimerHilo {
 public static void main(String[] args) {
     // Crear dos hilos
     MiHilo hilo1 = new MiHilo("Hilo 1");
     MiHilo hilo2 = new MiHilo("Hilo 2");

     // Iniciar los hilos
     hilo1.start();
     hilo2.start();
 }
}
```

Clase ***Main***:

- Aquí creamos dos instancias de ***MiHilo*** y llamamos al método ***start()*** en cada una.
- Esto inicia la ejecución de cada uno de los dos hilos.

4.1 La clase *Thread*

Ejecutar el programa

Al ejecutar el programa, verás que ambos hilos imprimen sus mensajes de forma concurrente, intercalando las salidas debido a la naturaleza asíncrona de los hilos.

4.1 La clase *Thread*

Ejemplo

Crea un programa que, mediante la creación de un único hilo, imprima un número de veces un mensaje “Mensaje 1”, “Mensaje 2” …

Ideas.

- Envía como parámetro al hilo el número de veces que quieres repetir el mensaje.
- ¿Dónde deberías incluir la sentencia para imprimir?
- ¿Cómo puedes hacer para que se imprima un número de veces un mensaje?

4.1 La clase *Thread*

```java
package ejemplos;

public class PrimerHilo extends Thread {
    private int x;

    PrimerHilo(int x) {
        this.x = x;
    }

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

    public static void main(String[] args) {
        PrimerHilo p = new PrimerHilo(10);
        p.start();
    }// main

}//PrimerHilo
```

4.1 La clase *Thread*

Ejemplo

Realiza un programa que cree tres hilos y que cada uno de ellos imprima 5 veces el mensaje:

“Hilo: “ + nombre del hilo + “ contador “ + número.

4.1 La clase *Thread*

```
package ejemplos;
public class HiloEjemplo1 extends Thread {
   // constructor
   public HiloEjemplo1(String nombre) {
       super(nombre);
       System.out.println("CREANDO HILO:" + super.getName());
   }

   // metodo run
   public void run() {
       for (int i=0; i<5; i++)
          System.out.println("Hilo: " + getName() + " contador =  " + i);
   }

   public static void main(String[] args) {
       HiloEjemplo1 h1 = new HiloEjemplo1("Hilo A");
       HiloEjemplo1 h2 = new HiloEjemplo1("Hilo B");
       HiloEjemplo1 h3 = new HiloEjemplo1("Hilo C");

       h1.start();
       h2.start();
       h3.start();

       System.out.println("3 HILOS INICIADOS...");
   }// main

}// HiloEjemplo1
```

PRINCIPALES MÉTODOS DE LA CLASE *THREAD*

4.1 Principales métodos de la clase *Thread*

Métodos de control de hilos

start():

- Este método inicia un nuevo hilo de ejecución.
- Para invocarlo, se llama al método ***run()*** en un hilo separado, lo que permite la ejecución concurrente.
- Una vez que un hilo se ha iniciado con ***start()***, no se puede volver a iniciar; en caso de hacerlo, tendremos una ***IllegalThreadStateException***.

run():

- Se define el código que se ejecutará cuando el hilo se inicie.
- Si no se sobrescribe el método, el hilo no realizará ninguna acción.
- El método ***run()*** contiene el trabajo que el hilo debe realizar, como cálculos, procesamiento de datos o interacciones de red.

join():

- Se utiliza para esperar a que un hilo específico termine su ejecucción.
- Es útil para garantizar que una tarea se complete antes de continuar con el siguiente paso.
- También puedes usar la sobrecarga ***join(long millis)*** para esperar un tiempo específico antes de continuar

4.1 Principales métodos de la clase *Thread*

Métodos de control de hilos

isAlive():

- Devuelve ***true*** si el hilo ha sido iniciado y aún está en ejecución.
- Esto es útil para verificar el estado de un hilo antes de realizar operaciones que dependen de su estado.
- Por ejemplo, puedes evitar llamar a ***join()*** en un hilo que ya ha terminado

setPriority(int priority):

- Establece la prioridad del hilo, entre ***Thread.MIN_PRIORITY (1)*** y ***Thread.MAX_PRIORITY (10)***.
- Influye en el tiempo de CPU que asigna el sistema operativo a los hilos.
- Su efectividad depende de la implementación del sistema operativo; algunos sistemas operativos no respetan la prioridad de los hilos de manera estricta.

getPriority():

- Este método devuelve la prioridad actual del hilo.
- Esto puede ser útil si necesitas ajustar el comportamiento del hilo en función de su prioridad o para registrar información sobre su configuración

4.1 Principales métodos de la clase *Thread*

Métodos de suspensión y reanudación

sleep(long millis):

- Este método hace que el hilo actual se suspenda durante un tiempo especificado en milisegundos.
- Es útil para crear pausas en la ejecución, por ejemplo, en operaciones repetitivas o al esperar eventos externos.
- Se puede interrumpir con ***interrupt()***, lo que lanzará una ***InterruptedException***.

yield():

- Cuando se llama a este método, el hilo actual sugiere al sistema operativo que ceda su tiempo de CPU a otros hilos que estén esperando.
- Sin embargo, no hay garantía de que esto suceda; es solo una sugerencia al programador del sistema operativo.

4.1 Principales métodos de la clase *Thread*

Métodos de interrupción

interrupt():

- Este método se utiliza para interrumpir un hilo.
- Si el hilo está bloqueado en métodos como ***sleep()*** o ***wait()***, se lanzará una ***InterruptedException***.
- Esto permite a los hilos responder a interrupciones y finalizar su trabajo de manera controlada.
- Es importante manejar esta excepción adecuadamente para liberar recursos.

4.1 Principales métodos de la clase *Thread*

Métodos de identificación y depuración

setName(String name):

- Permite establecer un nombre para el hilo, lo que puede ser muy útil para depurar, ya que puedes identificar fácilmente hilos en registros o herramientas de monitoreo.

getName():

- Devuelve el nombre asignado al hilo.
- Puedes usar este método para registrar información o realizar operaciones específicas basadas en el nombre del hilo

currentThread():

- Este método devuelve una referencia al hilo que está actualmente en ejecución.
- Es útil cuando necesitas realizar operaciones en el hilo actual sin mantener una referencia explícita, como establecer su nombre o su prioridad

4.1 Principales métodos de la clase *Thread*

Ejemplo

Veamos un ejemplo con alguno de los métodos anteriores:

- ***Thread.currentThread()*** devuelve una referencia al objeto hilo
- ***.setName ()***, ***.getName ()***
- ***.getPriority()***, ***setPriority()***.
- ***.toString()*** devuelve un String que representa al hilo: ***Thread [nombre del hilo, la prioridad, grupo de hilos]***
- ***.activeCount()*** devuelve el núm. hilos activos dentro del grupo.

4.1 Principales métodos de la clase *Thread*

```
public class MetodosThread {

    public static class HiloEjemplo extends Thread {

        public void run() {
            Thread hilo = Thread.currentThread();
            System.out.println("Dentro del Hilo : " + hilo.getName() + "\n\tPrioridad : " + hilo.getPriority());
        }
    }
    public static void main(String[] args) {
        Thread.currentThread().setName("Principal");
        System.out.println("Este hilo se llama " + Thread.currentThread().getName());
        System.out.println(Thread.currentThread().toString());

        HiloEjemplo HE = null;

        //Crea 3 hilos
        for (int i = 0; i < 3; i++) {
            HE = new HiloEjemplo(); //crear hilo
            HE.setName("HILO"+i); //damos nombre al hilo
            HE.setPriority(i+1); //damos prioridad
            HE.start(); //iniciar hilo
            System.out.println("Informacion del " + HE.getName() + ": "+ HE.toString());
        }
        System.out.println("3 HILOS CREADOS...");
        System.out.println("Hilos activos: " +Thread.activeCount());
    }

}
```

INTERFAZ *Runnable*

4.2 Interfaz *Runnable*

Para añadir la funcionalidad de hilo a una clase que deriva de otra clase (por ejemplo, un applet), 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 definimos la clase como

```
public class Reloj extends Applet implements Runnable()
```

4.2 Interfaz *Runnable*

La interfaz ***Runnable*** proporciona un único método, el método ***run()***.

La forma general de declarar un hilo implementando la intefaz ***Runnable*** es la siguiente:

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

4.2 Interfaz *Runnable*

Para crear un objeto hilo con el comportamiento de ***NombreHilo*** se escribe lo siguiente:

```
NombreHilo h=new NombreHilo();
```

Y para iniciar su ejecución utilizamos el método ***start()*** :

```
Thread h1= new Thread(h);
h1.start();
```

4.2 Interfaz *Runnable*

Ejemplo

Veamos un ejemplo con el funcionamiento de la interfaz ***Runnable***

4.2 Interfaz *Runnable*

Primero creamos la clase que implementa la interfaz ***Runnable***.

Contiene el método ***run()***, que es el código que se ejecutará cuando el hilo asociado a una instancia de esta clase se inicie

```
package Ejemplos;

class MiPrimerHiloR implements Runnable {

    public void run() {
        Thread hiloActual = Thread.currentThread ();
        System.out.println("Hola desde el Hilo " + hiloActual.getName());
    }
}
```

4.2 Interfaz *Runnable*

Esta es la clase que contiene el método ***main***, que es el punto de entrada de la aplicación.

Aquí se crean y se inician los hilos

```
package Ejemplos;

public class UsaMiPrimerHiloR {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //Primer hilo
        MiPrimerHiloR hilo1 = new MiPrimerHiloR();
        new Thread(hilo1).start();

        //Segundo hilo
        MiPrimerHiloR hilo2 = new MiPrimerHiloR();
        Thread hilo = new Thread(hilo2, "Hilo2");
        hilo.start();

        //Tercer Hilo
        new Thread(new MiPrimerHiloR()).start();
    }
}
```

4.2 Interfaz *Runnable*

- Se crea una instancia de ***MiPrimerHiloR***
- Luego, se crea un nuevo objeto ***Thread*** pasando ***hilo1*** como argumento y se llama a ***start()*** para iniciar la ejecución del hilo.

```
//Primer hilo
MiPrimerHiloR hilo1 = new MiPrimerHiloR();
new Thread(hilo1).start();
```

- Se crea otra instancia de ***MiPrimerHiloR***.
- Luego, se crea un nuevo ***hilo*** (llamado ***hilo***) y se le asigna el nombre "***Hilo2***".
- Llamamos a ***start()*** para ejecutar el hilo.

```
//Segundo hilo
MiPrimerHiloR hilo2 = new MiPrimerHiloR();
Thread hilo = new Thread(hilo2, "Hilo2");
hilo.start();
```

- Se crea una nueva instancia de ***MiPrimerHiloR*** y se pasa directamente al constructor de ***Thread***.
- Se llama a ***start()*** para ejecutar el hilo.

```
//Tercer Hilo
new Thread(new MiPrimerHiloR()).start();
```

4.2 Interfaz *Runnable*

Ejemplo Interfaz *Runnable*

Crea una clase que emule una caja en un supermercado, imprimiendo un mensaje diciendo que está atendiendo al cliente X en la caja Y, simule el proceso de pago esperando 5 segundos, y luego imprima un mensaje diciendo que el cliente X ha terminado en la caja Y.

Crea otras dos clases, cada una de ella debe emular la llegada de dos clientes. La primera debe emular que primero llega uno y luego otro, y la segunda que llegan los dos simultáneamente.

Ayuda. La llegada de un cliente significa la creación de un hilo, dos clientes → dos hilos.

4.2 Interfaz *Runnable*

Primero la clase que emula la caja de un supermercado

```
class CajaSupermercado implements Runnable {
    private String nombreCliente;

    public CajaSupermercado(String nombreCliente) {
        this.nombreCliente = nombreCliente;
    }

    @Override
    public void run() {
        System.out.println("El cliente " + nombreCliente + " está siendo atendido en "
                + Thread.currentThread().getName());
        try {
            // Simula el tiempo de atención (pausa de 5 segundos)
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("El cliente " + nombreCliente + " ha terminado su compra.");
    }
}
```

4.2 Interfaz *Runnable*

Ahora emulamos que llegan dos clientes, primero uno y después el otro

```java
public class Atender2Clientes {
    public static void main(String[] args) {
        // Crear instancias de CajaSupermercado para cada cliente
        CajaSupermercado cliente1 = new CajaSupermercado("Cliente 1");
        CajaSupermercado cliente2 = new CajaSupermercado("Cliente 2");

        Thread hilo1 = new Thread(cliente1, "caja1");
        Thread hilo2 = new Thread(cliente2, "caja2");

        // Llamar a los clientes uno tras otro
        hilo1.start();
        try {
            hilo1.join(); // Esperar a que termine el primer cliente
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        hilo2.start();

        // Esperar a que el segundo cliente termine
        try {
            hilo2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Ambos clientes han terminado sus compras.");
    }
}
```

4.2 Interfaz *Runnable*

Ahora emulamos que llegan dos clientes, de manera simultáneamente

```java
public class Atender2ClientesJuntos {

    public static void main(String[] args) {
        // Crear instancias de CajaSupermercado para cada cliente
        CajaSupermercado cliente1 = new CajaSupermercado("Cliente 1");
        CajaSupermercado cliente2 = new CajaSupermercado("Cliente 2");

        Thread hilo1 = new Thread(cliente1, "caja1");
        Thread hilo2 = new Thread(cliente2, "caja2");

        // Llamar a los clientes a la vez
        hilo1.start();
        hilo2.start();

        try {
            //Esperar a que terminen los dos clientes
            hilo1.join();
            hilo2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Ambos clientes han terminado sus compras.");
    }
}
```

4.2 Interfaz *Runnable*

Ejemplo Interfaz *Runnable*

¿Realmente hemos llamado a dos procesos simultáneos?

Comprobémoslo agregando funcionalidades a nuestro programa. El primer cliente compra 6 artículos que tardan 2, 2, 1, 5, 2 y 3 segundos respectivamente, el segundo cliente compra 5 artículos que tardan 1, 3, 5, 1 y 1 segundos respectivamente..

Modifica los mensajes para muestre el tiempo que tarda con cada cliente y producto, haciendo que el programa lleve el control de tiempos con el primer y segundo cliente.

4.2 Interfaz *Runnable*

Primero la clase que emula la caja de un supermercado

```
class CajaSupermercado implements Runnable {
    private String nombreCliente;
    private int[] tiemposProductos;
    private static int tiempoTotalGlobal = 0; // Tiempo total compartido entre todos los clientes

    public CajaSupermercado(String nombreCliente, int[] tiemposProductos) {
        this.nombreCliente = nombreCliente;
        this.tiemposProductos = tiemposProductos;
    }
    @Override
    public void run() {
        synchronized (CajaSupermercado.class) {
            System.out.println("Comienza " + nombreCliente + ", tiempo " + tiempoTotalGlobal
                    + " en " + Thread.currentThread().getName());
            for (int i = 0; i < tiemposProductos.length; i++) {
                int tiempoCompra = tiemposProductos[i] * 1000; // Convertir a milisegundos
                try {
                    Thread.sleep(tiempoCompra);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                // Actualizar tiempo total
                tiempoTotalGlobal += tiemposProductos[i];
                System.out.println("Vendido producto " + (i + 1) + " del " + nombreCliente
                        + " en " + Thread.currentThread().getName() + ", tiempo "
                        + tiempoTotalGlobal + " segundos");
            }
            System.out.println("El cliente " + nombreCliente + " ha terminado su compra en "
                    + Thread.currentThread().getName() + ", tiempo total " + tiempoTotalGlobal + " segundos.");
        }
    }
    // Método para obtener el tiempo total acumulado
    public static int getTiempoTotalGlobal() {
        return tiempoTotalGlobal;
    }
}
```

4.2 Interfaz *Runnable*

Ahora emulamos que llegan dos clientes, primero uno y después el otro

```
public class Atender2Clientes {

    public static void main(String[] args) {
        int[] tiemposCliente1 = {2, 2, 1, 5, 2, 3}; // Tiempos en segundos para el cliente 1
        int[] tiemposCliente2 = {1, 3, 5, 1, 1}; // Tiempos en segundos para el cliente 2
        // Crear instancias de CajaSupermercado para cada cliente
        CajaSupermercado cliente1 = new CajaSupermercado("Cliente 1", tiemposCliente1);
        CajaSupermercado cliente2 = new CajaSupermercado("Cliente 2", tiemposCliente2);

        Thread hilo1 = new Thread(cliente1, "caja1");
        Thread hilo2 = new Thread(cliente2, "caja2");

       // Llamar a los clientes uno tras otro
        hilo1.start();
        try {
            hilo1.join(); // Esperar a que termine el primer cliente
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        hilo2.start();
        // Esperar a que el segundo cliente termine
        try {
            hilo2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // Obtener el tiempo total acumulado
        int tiempoTotalFinal = CajaSupermercado.getTiempoTotalGlobal();
        System.out.println("Ambos clientes han terminado sus compras. Tiempo total acumulado: "
        + tiempoTotalFinal + " segundos.");
    }
}
```

4.2 Interfaz *Runnable*

Ahora emulamos que llegan dos clientes, de manera simultáneamente

```java
public class Atender2ClientesJuntos {

    public static void main(String[] args) {

        int[] tiemposCliente1 = {2, 2, 1, 5, 2, 3}; // Tiempos en segundos para el cliente 1
        int[] tiemposCliente2 = {1, 3, 5, 1, 1}; // Tiempos en segundos para el cliente 2

        // Crear instancias de CajaSupermercado para cada cliente
        CajaSupermercado cliente1 = new CajaSupermercado("Cliente 1", tiemposCliente1);
        CajaSupermercado cliente2 = new CajaSupermercado("Cliente 2", tiemposCliente2);

        Thread hilo1 = new Thread(cliente1, "caja1");
        Thread hilo2 = new Thread(cliente2, "caja2");

        // Iniciar hilos
        hilo1.start();
        hilo2.start();

        // Esperar a que ambos terminen
        try {
            hilo1.join();
            hilo2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Obtener el tiempo total acumulado
        int tiempoTotalFinal = CajaSupermercado.getTiempoTotalGlobal();
        System.out.println("Ambos clientes han terminado sus compras. Tiempo total acumulado: "
        + tiempoTotalFinal + " segundos.");
    }
}
```

4.2 Interfaz *Runnable*

Ejemplo Interfaz *Runnable*

Si ejecutamos ambos programas ¿Qué ocurre? ¿Hemos calculado bien los tiempos?

Piénsalo y ponlo en práctica en la siguiente práctica.