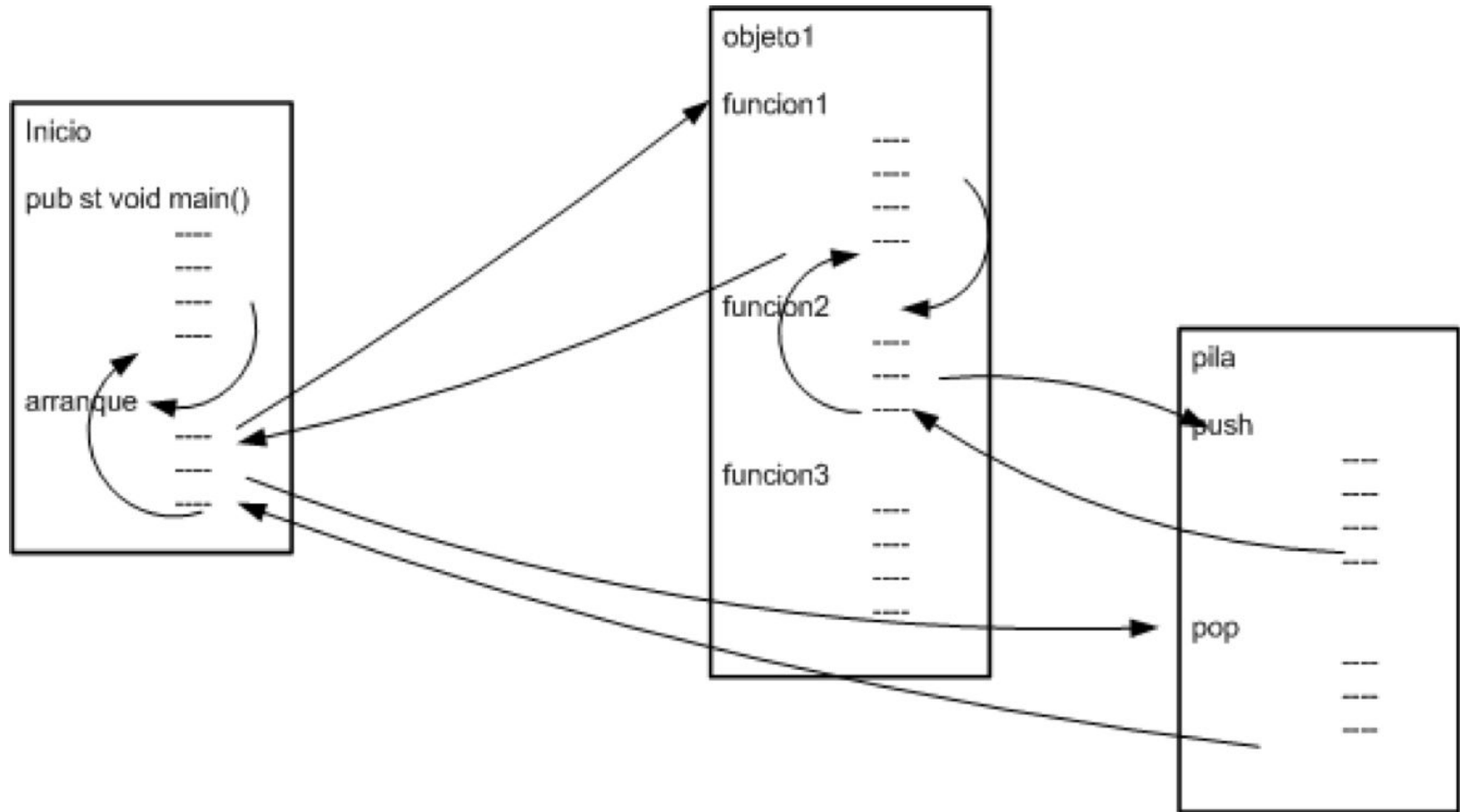


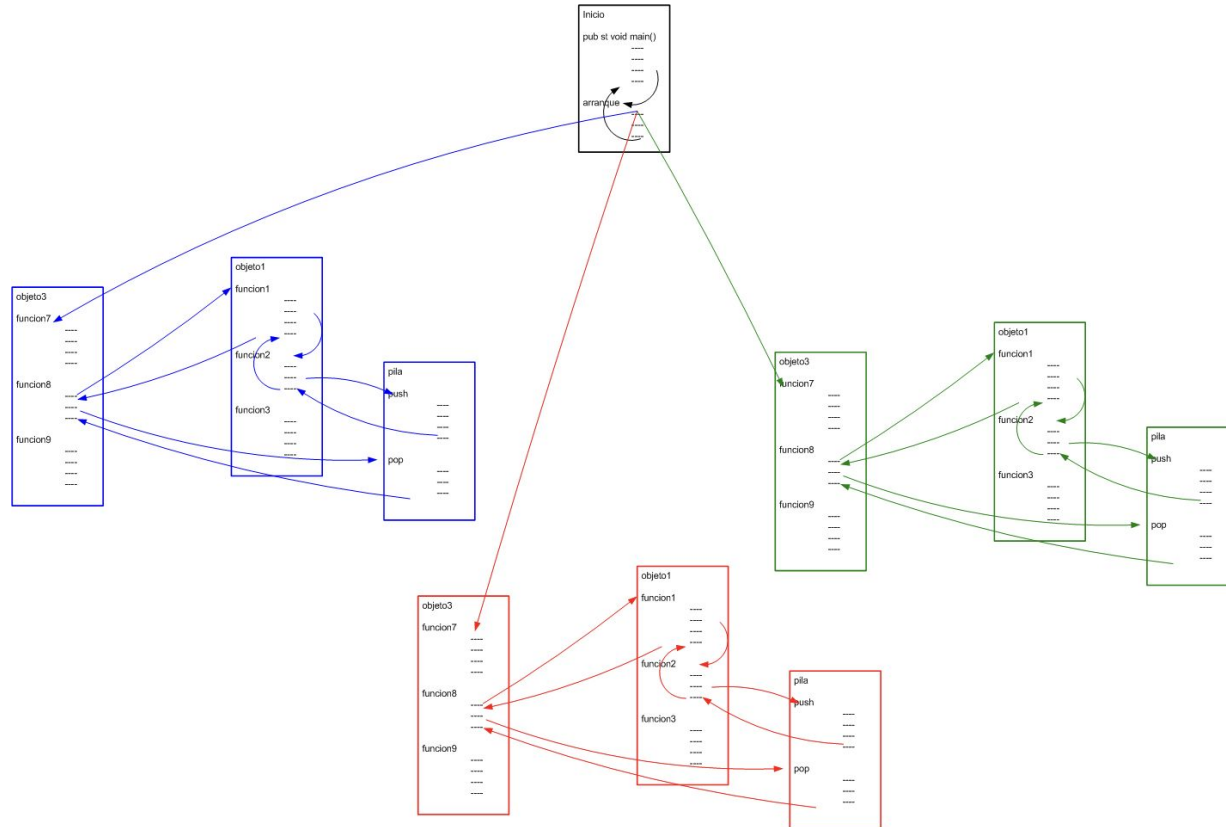
Único camino de ejecución



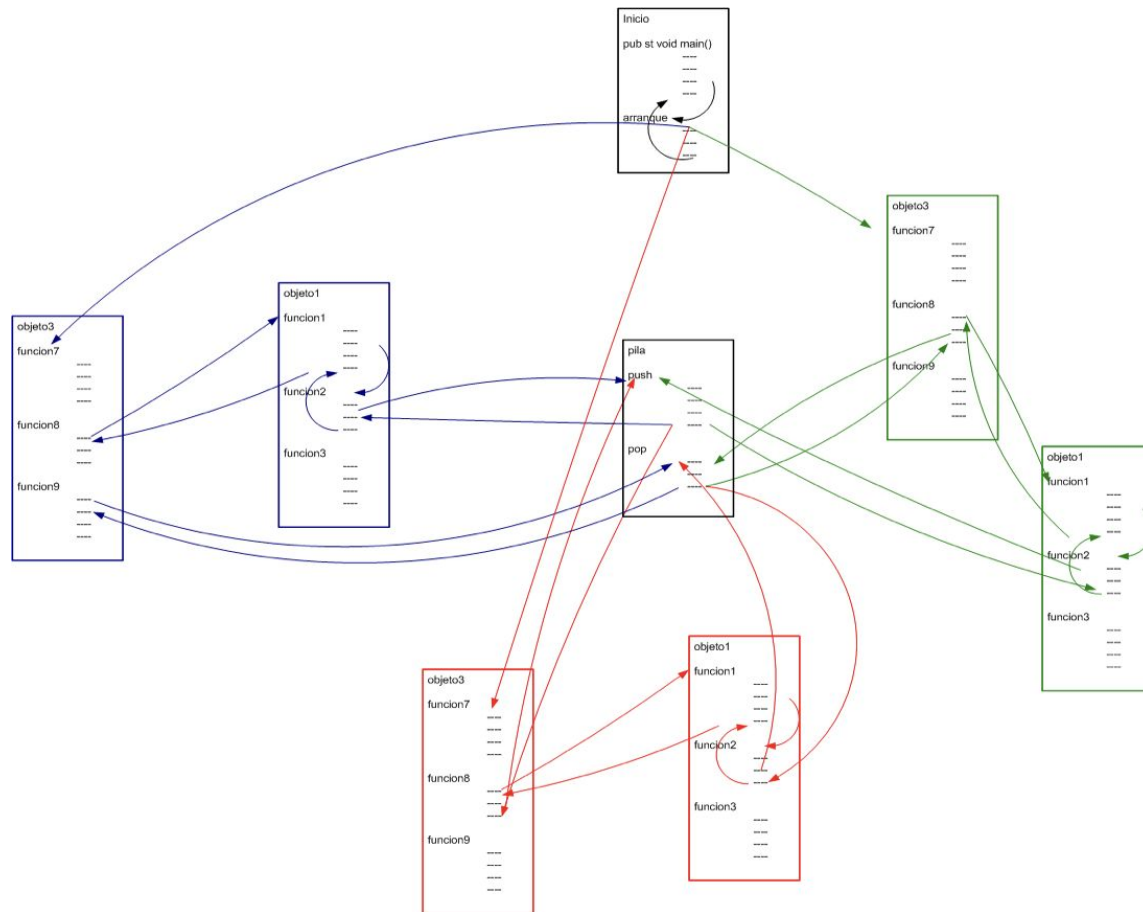
Clase 6

Hilos

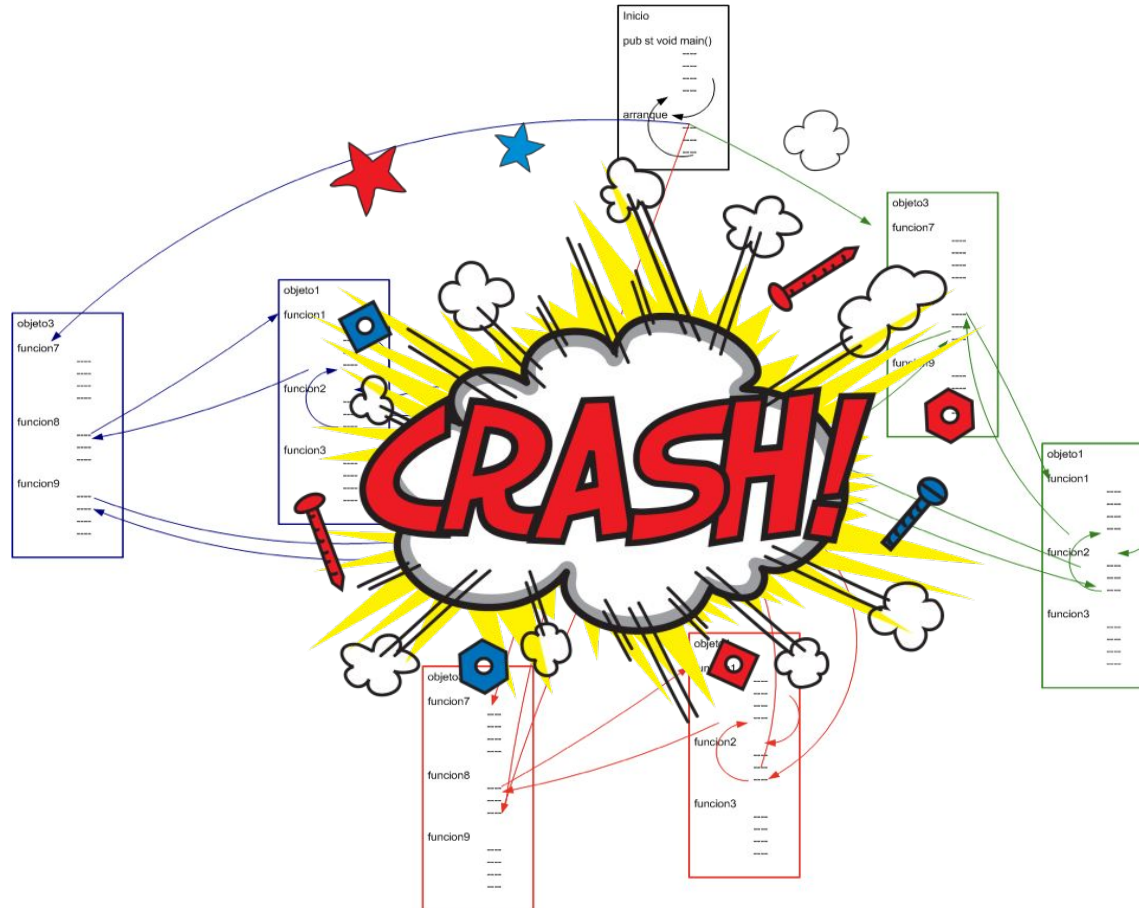
Hilos



Consistencia de datos cuando hay concurrencia



Consistencia de datos cuando hay concurrencia



Solución al problema de consistencia

Alternativas

- Bloqueo sincronizado
- Utilizar colecciones concurrentes
- Evitar variables estáticas
- Patrones de diseño (productor-consumidor, pool de datos)

Solución al problema de consistencia

Bloqueo sincronizado

- Se usa la palabra reservada synchronized
- Utiliza una llave (es un objeto)
- Sincronización de métodos o bloques de código

```
public class ClaseX {  
    public boolean llamado1(String dato) {}  
    public boolean llamado2(String dato) {}  
    public boolean llamado3(String dato) {}  
    public boolean llamado4(String dato) {}  
}
```

Bloqueo sincronizado

```
public boolean llamado1(String dato) {  
    String var1 = "var1";  
    String var2 = "var2";  
    String var3 = "var3";  
    String var4 = "var4";  
    String var5 = "var5";  
    String var6 = "var6";  
    return true;  
}
```

```
public synchronized boolean llamado1(String dato) {  
    String var1 = "var1";  
    String var2 = "var2";  
    String var3 = "var3";  
    String var4 = "var4";  
    String var5 = "var5";  
    String var6 = "var6";  
    return true;  
}
```


Bloqueo sincronizado

```
public boolean llamado2(String dato) {  
    String var1 = "var1";  
    String var2 = "var2";  
    synchronized (this.llave) {  
        String var3 = "var3";  
        String var4 = "var4";  
    }  
    String var5 = "var5";  
    String var6 = "var6";  
    return true;  
}
```

```
public boolean llamado4(String dato) {  
    String var1 = "var1";  
    String var2 = "var2";  
    synchronized (this) {  
        String var3 = "var3";  
        String var4 = "var4";  
    }  
    String var5 = "var5";  
    String var6 = "var6";  
    return true;  
}
```

Solución al problema de consistencia

Alternativas

Utilizar colecciones concurrentes (thread safe)

Evitar variables estáticas

Patrones de diseño (pool de datos, productor-consumidor)

Productor - Consumidor

Características

- Múltiples hilos interactuando con un recurso común (almacenamiento)
- Productor: responsable de agregar datos al almacenamiento
- Consumidor: responsable de sacar datos del almacenamiento
- Es asíncrono
- El productor y consumidor se notifican de alguna forma (wait, notify, notifyAll)

Productor - Consumidor

```
public class Almacenamiento {
    private Queue<Integer> cola;
    private int capacidad;

    public Almacenamiento () {
        cola = new LinkedList<>();
        capacidad = 5;
    }

    public synchronized void agregar(int valor)
    {
        while (cola.size() == capacidad) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        cola.add(valor);
        System.out.println("Productor agrega "
+ valor + " en la cola.");
        notifyAll();
    }
}
```

```
    public synchronized int retirar() {
        while (cola.isEmpty()) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        int valor = cola.remove();
        System.out.println("Consumidor retira "
+ valor + " de la cola.");
        notifyAll();
        return valor;
    }
}
```

Productor - Consumidor

```
class Productor implements Runnable {
    private Almacenamiento almacenamiento;

    public Productor(Almacenamiento almacenamiento) {
        this.almacenamiento = almacenamiento;
    }

    public void run() {
        for (int i = 1; i <= 10; i++) {
            almacenamiento.agregar(i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Productor - Consumidor

```
class Consumidor implements Runnable {
    private Almacenamiento almacenamiento;

    public Consumidor(Almacenamiento almacenamiento) {
        this.almacenamiento = almacenamiento;
    }

    public void run() {
        for (int i = 1; i <= 10; i++) {
            int valor = almacenamiento.retirar();
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Productor - Consumidor

```
public class ArranquePC {

    public static void main(String[] args) {
        ArranquePC arranque = new ArranquePC();
        arranque.arranque();
    }

    public void arranque() {
        // Creamos una instancia del almacenamiento compartido
        Almacenamiento almacenamiento = new Almacenamiento();

        // Creamos una instancia del productor y lo iniciamos
        Productor productor = new Productor(almacenamiento);
        Thread hiloProductor = new Thread(productor);
        hiloProductor.start();

        // Creamos una instancia del consumidor y lo iniciamos
        Consumidor consumidor = new Consumidor(almacenamiento);
        Thread hiloConsumidor = new Thread(consumidor);
        hiloConsumidor.start();
    }
}
```

Definición de hilos

Los hilos se pueden generar de 3 formas

Forma “original”

- Extendiendo la clase Thread
- Implementando la interfaz Runnable

Forma más reciente

- Implementando la interfaz Callable
 - Implementa genérico Callable<T>
 - Puede devolver un valor usando la clase Future y el método get()
 - Ejecución usando ExecutorService

Definición de hilos - Callable

```
public class HiloCallable implements Callable<Void> {
    protected String nombre;
    protected int demora;
    protected int iteraciones;

    public HiloCallable() {}

    public HiloCallable(String nombre, int demora, int iteraciones) {
        this.nombre = nombre; this.demora = demora; this.iteraciones = iteraciones;
    }

    @Override
    public Void call() throws Exception {
        System.out.println("Arranca el hilo");
        for(int i=0; i<this.iteraciones; i++) {
            StringBuffer sb = new StringBuffer();
            sb.append("Hilo nombre: "+this.nombre);
            sb.append(" - iteracion: "+i+" de "+this.iteraciones);
            System.out.println(sb.toString());
            try {
                Thread.sleep(this.demora);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
        return null;
    }
}
```

Definición de hilos - Callable

```
public class HiloCallable2 implements Callable<String> {
    protected String nombre; protected int demora;
    protected int iteraciones; protected boolean mostrar;

    public HiloCallable2 (String nombre, int demora, int iteraciones) {
        this.nombre = nombre; this.demora = demora; this.iteraciones = iteraciones;
        this.mostrar = true;
    }

    @Override
    public String call() throws Exception {
        System.out.println("Arranca el hilo: "+this.nombre);
        for(int i=0; i<this.iteraciones; i++) {
            StringBuffer sb = new StringBuffer();
            sb.append("Hilo nombre: "+this.nombre);
            sb.append(" - iteracion: "+i+" de "+this.iteraciones);
            if(this.mostrar) {
                System.out.println(sb.toString());
            }
            try {
                Thread.sleep(this.demora);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
        return "hilo "+this.nombre+" finalizado";
    }
}
```

ExecutorService

Características

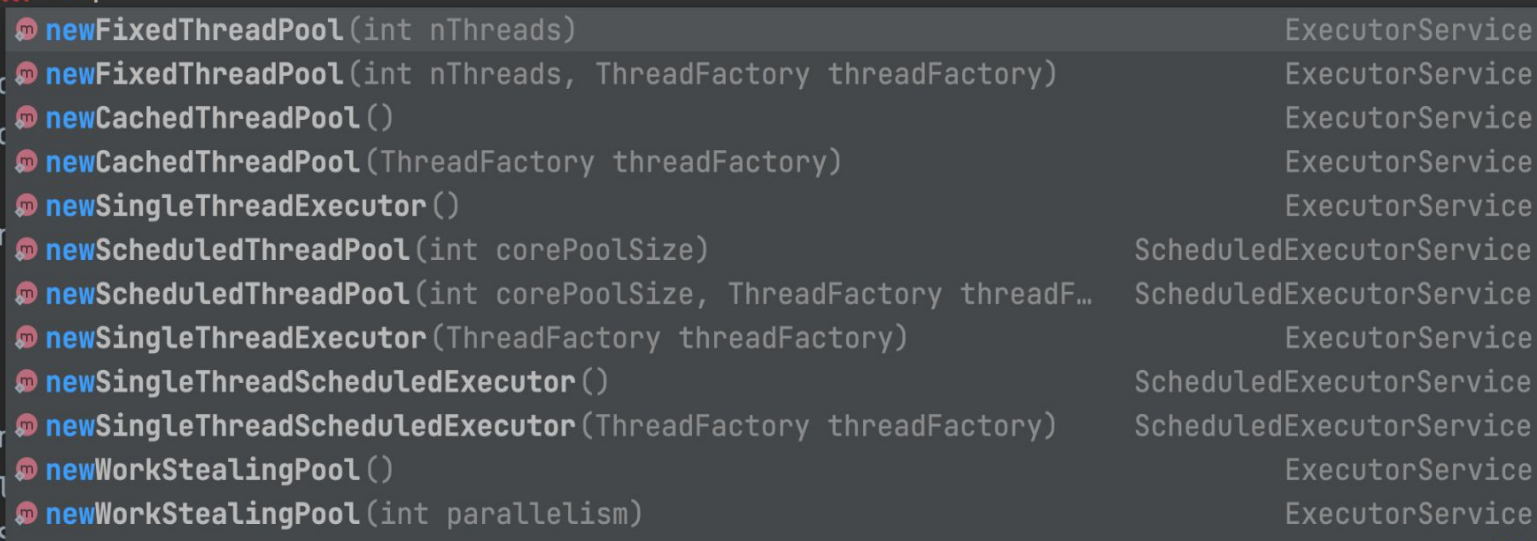
- Ejecutor de hilos (asíncrono y paralelo)
- Maneja la creación, gestión y terminación de hilos.
- Más eficiente y controlable que el manejo de Thread nativo.
- Maneja pool de hilos.
- Ejecución única o repetitiva.
- Varias implementaciones de ES de acuerdo a nuestras necesidades.
- Timeout en la ejecución (get)

ExecutorService - ejecución

```
public void arranque() {  
    ExecutorService executorService = Executors.newFixedThreadPool(5);  
    for(int i = 0; i < 10; i++) {  
        HiloCallable hc = new HiloCallable("hilo: "+i,100,10);  
        executorService.submit(hc);  
    }  
    executorService.shutdown();  
}
```

ExecutorService - tipos

```
public void arranque() {  
    ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 5);  
    Executors.new  
    for(int i = 0; i < nThreads; i++) {  
        Hilo h = new Hilo(i, executorService);  
        executorService.execute(h);  
    }  
    executorService.shutdown();  
}  
  
public void mostrarResultado() {  
    ExecutorService executorService = Executors.newSingleThreadExecutor();  
    HiloCall h = new HiloCall(executorService);  
    Future<String> resultado = h.call();  
    while(!resultado.isDone()) {  
        try {  
            resultado.get();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```



Method Signature	Return Type
<code>newFixedThreadPool(int nThreads)</code>	<code>ExecutorService</code>
<code>newFixedThreadPool(int nThreads, ThreadFactory threadFactory)</code>	<code>ExecutorService</code>
<code>newCachedThreadPool()</code>	<code>ExecutorService</code>
<code>newCachedThreadPool(ThreadFactory threadFactory)</code>	<code>ExecutorService</code>
<code>newSingleThreadExecutor()</code>	<code>ExecutorService</code>
<code>newScheduledThreadPool(int corePoolSize)</code>	<code>ScheduledExecutorService</code>
<code>newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)</code>	<code>ScheduledExecutorService</code>
<code>newSingleThreadExecutor(ThreadFactory threadFactory)</code>	<code>ExecutorService</code>
<code>newSingleThreadScheduledExecutor()</code>	<code>ScheduledExecutorService</code>
<code>newSingleThreadScheduledExecutor(ThreadFactory threadFactory)</code>	<code>ScheduledExecutorService</code>
<code>newWorkStealingPool()</code>	<code>ExecutorService</code>
<code>newWorkStealingPool(int parallelism)</code>	<code>ExecutorService</code>

Press ^, to choose the selected (or first) suggestion and insert a dot afterwards [Next Tip](#)

ExecutorService - leer resultado

```
public void arranque2() {
    ExecutorService executorService = Executors.newFixedThreadPool(1);
    HiloCallable2 hc = new HiloCallable2("hilo1", 1500, 50);
    Future<String> resultado = executorService.submit(hc);
    while(resultado.isDone()) {

        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
    try {
        System.out.println("Resultado de la ejecución: "+resultado.get());
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } catch (ExecutionException e) {
        throw new RuntimeException(e);
    }
    executorService.shutdown();
}
```

ExecutorService - leer resultado

```
public void arranque3() {  
    ExecutorService executorService = Executors.newFixedThreadPool(5);  
    List<Future> resultados = new ArrayList<>();  
    int iteraciones = 10;  
    for(int i = 0; i < iteraciones; i++) {  
        Random aleatorio = new Random(System.currentTimeMillis());  
        HiloCallable2 hc = new HiloCallable2("hilo: "+i,aleatorio.nextInt(900)+100,  
            aleatorio.nextInt(5)+10);  
        hc.setMostrar(false);  
        Future<String> resultado = executorService.submit(hc);  
        resultados.add(resultado);  
    }  
}
```

ExecutorService - leer resultado

```
boolean continuar = true;
while(continuar) {
    if(resultados.size()==0) {
        continuar=false;
    }
    for(int i = 0; i < resultados.size(); i++) {
        if(resultados.get(i).isDone()) {
            try {
                System.out.println("Resultado de ejecución: "+resultados.get(i).get());
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
            resultados.remove(i);
        }
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
executorService.shutdown();
}
```