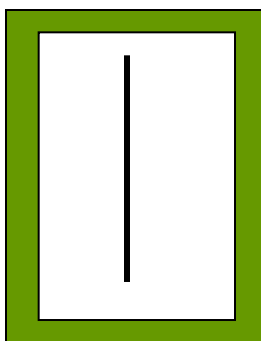


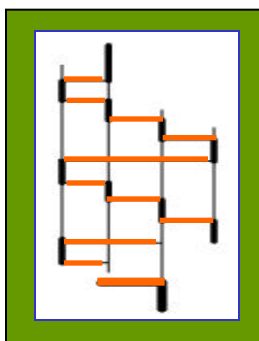
# Concurrencia Threads

# Threads

- Un **thread** es un flujo de control secuencial dentro de un proceso. A los threads también se los conoce como **procesos livianos** (requiere menos recursos crear un thread nuevo que un proceso nuevo) ó **contextos de ejecución**.
- Un **thread** es similar a un programa secuencial: tiene un comienzo, una secuencia de ejecución, un final y en un instante de tiempo dado hay un único punto de ejecución. Sin embargo, un thread no es un programa. Un **thread** se ejecuta adentro de un programa.
- Lo novedoso en **threads** es el uso de múltiples threads adentro de un mismo programa, ejecutándose simultáneamente y realizando tareas diferentes:



Programa *singleThread*



Programa *multiThread*

Thread en ejecución  
Transferencia del control  
Thread *bloqueado*



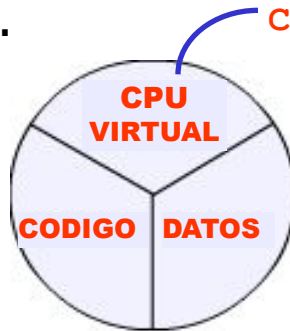
En un programa *multithread*, más de un thread se ejecuta en forma concurrente. El control de ejecución es transferido entre los diferentes threads, cada uno de los cuáles es responsable de distintas tareas.

- En el modelo de **multithreading** la CPU asigna a cada thread un tiempo para que se ejecute; cada thread “tiene la percepción” que dispone de la CPU constantemente, sin embargo el tiempo de CPU está dividido entre todos los threads.

# Threads

- Un **thread** se ejecuta dentro del contexto de un programa o proceso y comparte los recursos asignados al programa. A pesar de esto, los **threads** toman algunos recursos del ambiente de ejecución del programa como propios: tienen su propia pila de ejecución, contador de programa, código y datos. Como un **thread** solamente se ejecuta dentro de un contexto, a un thread también se lo llama **contexto de ejecución**.
- Kotlin y por ende JAVA soporta programas **multithreading** a través del lenguaje, de librerías y del sistema de ejecución.

Múltiples-threads comparten el mismo código, cuando se ejecutan a partir de instancias de la misma clase.



CPU virtual que ejecuta código y utiliza datos

Múltiples-threads comparten datos, cuando acceden a objetos comunes (podría ser a partir de códigos diferentes).

- La **funcion thread()** proporcionada por el paquete *kotlin.concurrent* que facilita la creación y ejecución de hilos en un programa.
- Hay dos estrategias para usar **Threads**:
  - **Directamente controlar la creación y el gerenciamiento** instanciando un Thread cada vez que la aplicación requiere iniciar una tarea concurrente.
  - **Abstraer el gerenciamiento** de threads pasando la tarea concurrente a un **ejecutor** para que la administre y ejecute.

# Creación y Gerenciamiento de Threads

- La **funcion thread()** provee el comportamiento genérico de los threads Kotlin: arranque, ejecución, interrupción, asignación de prioridades, etc.
- La **funcion thread()** tiene la siguiente firma:

```
fun thread(start: Boolean = true, isDaemon: Boolean = false, contextClassLoader: ClassLoader? = null, block: () -> Unit): Thread
```

- **start:** Indica si el hilo debe comenzar a ejecutarse inmediatamente después de ser creado. Por defecto, es true.
  - **isDaemon:** Indica si el hilo debe ser un hilo daemon y estos no impiden que la JVM se cierre. Por defecto, es false.
  - **contextClassLoader:** Un ClassLoader que se establece como el cargador de clases del nuevo hilo. Puede ser null si se quiere utilizar el cargador de clases actual.
  - **block:** Es un bloque lambda que contiene el código que se ejecutará en el nuevo hilo.
  - La función devuelve una instancia de Thread que referencia al nuevo hilo creado.
- Tanto Kotlin como JAVA son **multithread**: siempre hay un thread ejecutándose junto con las aplicaciones de los usuarios, por ejemplo el **garbage collector** es un thread que se ejecuta en background; las GUI's **dibujan las componentes** en la pantallas, etc.

# Ejemplo del uso de thread()

Es un método estándar de la clase **Thread**. Es el lugar donde el **thread** comienza su ejecución.

```
import kotlin.concurrent.thread

fun main(args: Array<String>) {

    for(i in 1 .. 5) {
        thread() {
            val name="thread_${i}"
            var contador = 10
            for (j in 1..contador) {
                println("#${name}:${j}")
            }
        }
    }
}
```

Inicializa el objeto Thread y el thread pasa a estado "vivo"

Cuando finaliza el for, hay 6 threads ejecutándose en paralelo: el thread que invocó a la thread(), en nuestro caso el main thread y los threads que están ejecutando los bloques lambda.

Un thread termina cuando finaliza el bloque lambda.

Tenemos 5 tareas concurrentes, cada una de ellas imprime en pantalla 10 veces su nombre. Además tenemos el **main thread**.

## ¿Cuál es la salida del programa?

La salida de una ejecución del programa podría ser diferente a la salida de otra ejecución del mismo programa, dado que el mecanismo de **scheduling** de threads no es determinístico.

# Sleep

# Métodos de la clase Thread

**Suspende** temporalmente la ejecución del **thread** que se está ejecutando. Afecta solamente al **thread** que ejecuta el **sleep()**, no es posible decirle a otro thread que "se duerma".

```
import java.util.concurrent.TimeUnit
import kotlin.concurrent.thread

fun main(args: Array<String>) {

    for(i in 1 .. 5) {

        thread() {
            val name="thread_${i}"
            var contador = 10
            for (j in 1..contador) {
                println("#${name}:${j}")
                //Antes de JSE 5:
                //Thread.sleep(100)
                //Estilo sugerido:
                TimeUnit.MILLISECONDS.sleep(100)
            }
        }
    }
}
```

- Thread.sleep()** es un método de clase que detiene la ejecución del hilo actual durante el tiempo especificado en milisegundos, el cual se pasa como parámetro.
- TimeUnit.MILLISECONDS.sleep()** es utilizado para detener la ejecución del hilo actual, es un método de la clase **TimeUnit**, que es una enumeración utilizada para representar distintas unidades de tiempo, como milisegundos, segundos y minutos.
- Los threads se ejecutan en cualquier orden. El método **sleep()** no permite controlar el orden de ejecución de los threads; suspende la ejecución del thread por un tiempo dado.
- En nuestro ejemplo, la única garantía que se tiene es que el thread suspenderá su ejecución por al menos 100 milisegundos, aunque podría tardar más en reanudar la ejecución. Además, existe la posibilidad de que se lance una excepción **InterruptedException**.

# Join Métodos de la clase Thread

El método **join()** permite que un **thread** espere a que otro termine de ejecutarse. El objetivo del método **join()** es esperar por un evento específico: la terminación de un **thread**. El **thread** que invoca al **join()** sobre otro **thread** se bloquea hasta que dicho **thread** termine. Una vez que el **thread** se completa, el método **join()** retorna inmediatamente.

```
import java.util.concurrent.TimeUnit
import kotlin.concurrent.thread

fun main(args: Array<String>) {
    val t = thread() {
        val name = "thread_1"
        var contador = 10
        for (j in 1..contador) {
            println("#${name}: ${j}")
            TimeUnit.MILLISECONDS.sleep(100)
        }
    }
    while (t.isAlive) {
        println("Esperando...")
        t.join()
    }
    println("Finalizado...")
}
```

```
while (t.isAlive()) {
    t.join(2000);
    print(" .");
}
println(" Finalizado!");
```

Este segmento de código inicia al thread **t** y cada 2 segundos imprime un "." mientras **t** continúa ejecutándose

El main thread se suspende hasta que el thread **t** termine (**isAlive()** devuelve false)

- En este caso el **main thread** se bloquea en espera que el **thread t** termine de ejecutarse.
- El método **join()** es sobrecargado, permite especificar el tiempo de espera. Sin embargo, de la misma manera que el **sleep()**, no se puede asumir que este tiempo sea preciso. Como el método **sleep()**, el **join()** responde a una interrupción terminando con una **InterruptedException**

# Métodos de la clase Thread

## Yield

Permite indicar al mecanismo de scheduling que el thread ya hizo suficiente trabajo y que podría cederle tiempo de CPU a otro thread. Su efecto es dependiente del SO sobre el que se ejecuta la JVM. Permite implementar **multithreading cooperativo**. Es una pista para el scheduling.

```
import kotlin.concurrent.thread

fun main(args: Array<String>) {

    for(i in 1 .. 5) {
        thread() {
            val name="thread_${i}"
            var contador = 10
            for (j in 1..contador) {
                println("#${name}:${j}")
                Thread.yield()
            }
        }
    }
}
```

El thread de esta manera realizaría un procesamiento mejor distribuido entre varias tareas del mismo tipo.



# Métodos de la clase Thread

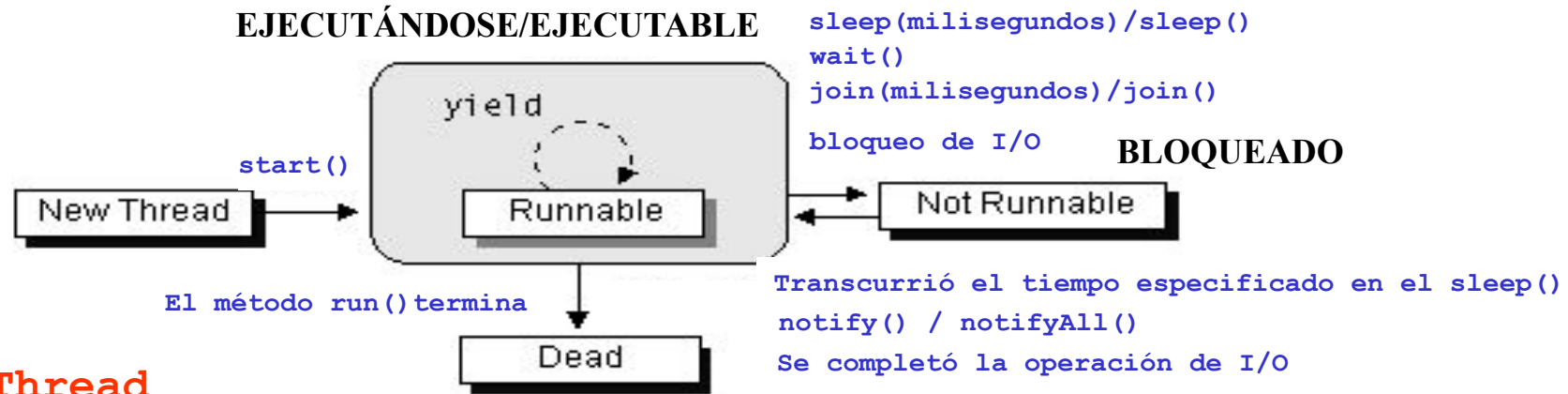
## Interrupt

Es un **pedido de interrupción**. El thread que lo recibe se interrumpe a sí mismo de una manera conveniente. El pedido causa que los métodos de bloqueo (**sleep()**, **join()**, **wait()**) disparen la excepción `InterruptedException` y además setea un *flag* en el **thread** que indica que al thread se le ha pedido que se interrumpa. Se usa el método **isInterrupted()** para preguntar por este *flag*.

```
import java.util.concurrent.TimeUnit
import kotlin.concurrent.thread

fun main(args: Array<String>) {
    val t = thread() {
        for (i in 1..10) {
            println(i)
            try {
                TimeUnit.MILLISECONDS.sleep(4000)
            } catch (e: InterruptedException) {
                println("El thread ${Thread.currentThread()} no puede terminar")
            }
        }
        println("Termino!")
    }
    val startTime = System.currentTimeMillis()
    while (t.isAlive) {
        println("Esperando.....")
        t.join(1000)
        if ((System.currentTimeMillis() - startTime) > 1000 * 60 * 60 && t.isAlive) {
            println("Cansado de esperar!")
            t.interrupt()
            t.join()
        }
    }
    println("Fin!")
}
```

# Ciclo de vida de un Thread



## Estado New Thread

Inmediatamente después que un thread es creado pasa a estado **New Thread**, pero aún no ha sido iniciado, por lo tanto no puede ejecutarse. Se debe invocar al método **start()**.

## Estado Running (Ejecutándose)/Runnable (Ejecutable)

Después de ejecutarse el método **start()** el thread pasa al estado **Runnable o Ejecutable**. Un thread arrancado con **start()** podría o no comenzar a ejecutarse. No hay nada que evite que el thread se ejecute. La JVM implementa una estrategia (scheduling) que permite compartir la CPU entre todos los threads en estado Runnable.

## Estado Not Runnable o Blocked (Bloqueado)

Un thread pasa a estado **Not Runnable o Bloqueado** cuando ocurren algunos de los siguientes eventos: se invoca al método **sleep()**, al **wait()**, **join()** ó **el thread está bloqueado en espera de una operación de I/O, el thread invoca a un método synchronized sobre un objeto y el lock del objeto no está disponible**. Cada entrada al estado **Not Runnable** tiene una forma de salida correspondiente. Cuando un thread está en estado bloqueado, el *scheduler* lo saltea y no le da ningún *slice* de CPU para ejecutarse.

## Estado Dead

Los **threads** definen su finalización implementando un **run()** que termine naturalmente.

# Ejemplo

```
import ...
fun main() {
    SwingUtilities.invokeLater {
        var fin = false
        val label = JLabel("00:00:00")
        val buttonStart = JButton("Arrancar")
        buttonStart.addActionListener {
            val relojThread = thread() {

                val formato = DateTimeFormatter.ofPattern("HH:mm:ss")
                while (!fin) {
                    label.text=LocalTime.now().format(formato)
                    TimeUnit.SECONDS.sleep(1)
                }
                label.text="00:00:00"
            }
        }

        val buttonStop = JButton("Parar")
        buttonStop.addActionListener {
            fin=true
        }

        val frame = JFrame("Reloj")
        frame.defaultCloseOperation = JFrame.EXIT_ON_CLOSE
        frame.setSize(300, 200)
        frame.contentPane.layout=FlowLayout()
        frame.contentPane.add(label)
        frame.contentPane.add(buttonStart)
        frame.contentPane.add(buttonStop)
        frame.isVisible = true
    }
}

} // Fin de la clase Reloj
```

Se crea una instancia de Thread, **relojThread**.  
Estado **NEW THREAD**

Crea los recursos para ejecutar el thread, organiza la ejecución del thread y **comienza la ejecución del while**.  
Estado **RUNNABLE**

Durante un segundo el thread está en estado **NOT RUNNABLE**

Esta asignación hace que la condición de continuación del while deje de cumplirse y de esta manera el thread finaliza. Pasa a estado **DEAD**

# Prioridades en Threads

- En las configuraciones de computadoras en las que se dispone de una única CPU, los threads se ejecutarán de a uno a la vez simulando concurrencia. Uno de los principales beneficios del modelo de **threading** es que permite abstraernos de la configuración de procesadores.
- Cuando múltiples threads quieren ejecutarse, es el **SO el que determina a cuál de ellos le asignará CPU**. Los **programas Kotlin pueden influir**, sin embargo la **decisión final es del SO**.
- Se llama **scheduling** a la estrategia que determina el orden de ejecución de múltiples threads sobre una única CPU.
- La **JVM soporta** un algoritmo de **scheduling simple** llamado **scheduling de prioridad fija**, que consiste en determinar el orden en que se ejecutarán los threads de acuerdo a la prioridad que ellos tienen.
- La prioridad de un thread le indica al **scheduler** cuán importante es.
- Cuando se crea un thread, éste hereda la prioridad del thread que lo creó (NORM\_PRIORITY ). Es posible modificar la prioridad de un thread después de su creación usando el método **setPriority(int)**. Las prioridades de los threads son números enteros que varían entre las constantes MIN\_PRIORITY y MAX\_PRIORITY (definidas en la clase Thread).

# Prioridades en Threads

- El sistema de ejecución elige para ejecutar entre los threads que están en estado **Runnable** aquel que tiene prioridad más alta. Cuando éste thread finaliza, cede el procesador o pasa a estado **Bloqueado**, comienza a ejecutarse un thread de más baja prioridad.
- El **scheduler** usa una estrategia **round-robin** para elegir entre dos threads de igual prioridad que están esperando por la CPU. El thread elegido se ejecuta hasta que un thread de más alta prioridad pase a estado **Runnable**, ceda la CPU a otro thread, finalice su ejecución ó, expire el tiempo de CPU asignado (time-slicing). Luego, el segundo thread tiene la posibilidad de ejecutarse.
- El algoritmo de **scheduling** también es **preemptive**: cada vez que un thread con mayor prioridad que todos los threads que están en estado **Runnable** pasa a estado **Runnable**, el sistema de ejecución elige el nuevo thread de mayor prioridad para ejecutarse.

# Ejecutores

Los **Ejecutores** simplifican la programación concurrente. Se incorporaron en JSE 5.

- Los **EJECUTORES** proveen una capa de indirección entre un cliente y la ejecución de una tarea. Es un objeto intermedio que ejecuta la tarea, desligando al cliente de la ejecución de la misma.
- Los **EJECUTORES** son objetos que encapsulan la creación y administración de **threads**, permitiendo **desacoplar** la tarea concurrente del mecanismo de ejecución. Entre sus responsabilidades están la creación, el uso y el *scheduling* de threads.
- Los **EJECUTORES** permiten modelar programas como una serie de tareas concurrentes asincrónicas, evitando los detalles asociados con **threads**: simplemente se crea una tarea que se pasa al ejecutor apropiado para que la ejecute.
- Un **EJECUTOR** es normalmente usado en vez de crear explícitamente **threads**:

Con threads creados por el programador:

```
val t = thread() { . . . }
```

Con EJECUTORES:

```
val e = unEjecutor  
e.execute { . . . }
```

- Un **EJECUTOR** es un objeto que implementa la interface **Executor**.

```
package java.util.concurrent;  
public interface Executor {  
    public void execute();  
}
```

Construye el contexto apropiado para ejecutar objetos Runnable

# Ejecutores

Con threads creados por el programador:

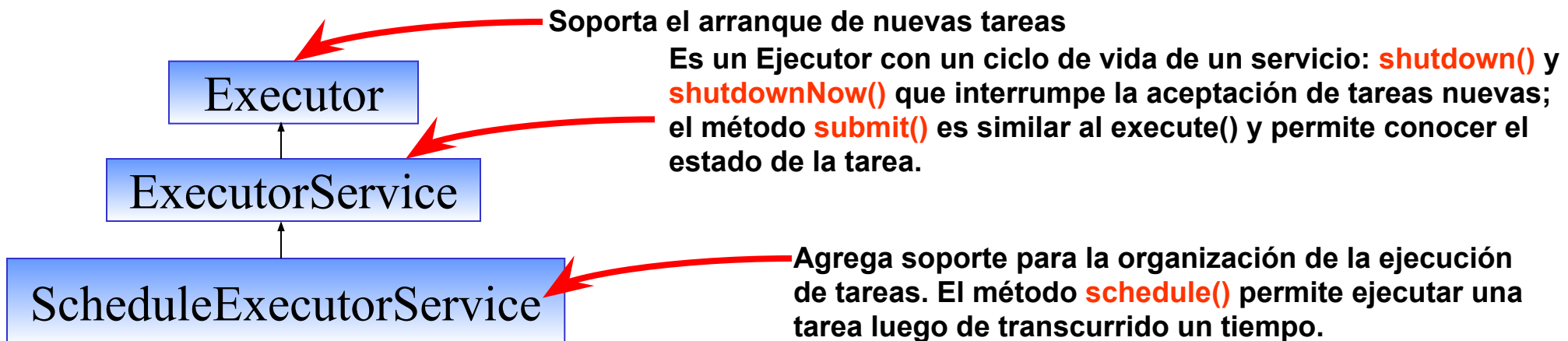
```
val t = thread() { . . . }
```

Con EJECUTORES:

```
val e = unEjecutor  
e.execute { . . . }
```

El comportamiento del método **execute()** es menos específico que el usado con **Threads**, siendo los **threads** creados y lanzamos inmediatamente. Dependiendo de la implementación del **Executor** el método **execute()** podría hacer lo mismo, o usar un **thread** existente disponible para ejecutar la tarea **r** o encolar **r** hasta que haya un **thread** disponible para ejecutar la tarea.

El paquete **java.util.concurrent** define tres interfaces Executor:



# Ejecutores & Pool de Threads

Típicamente las implementaciones de **EJECUTORES** del paquete **java.util.concurrent** usan *pool de threads*. Estos threads existen independientemente de las tareas Runnablees que ejecutan y generalmente ejecutan múltiples tareas.

El pool de threads minimiza la sobrecarga causada por la creación de nuevos threads=> **reuso de threads.**

Aumenta la *performance* de aplicaciones que ejecutan muchos **threads** simultáneamente. El pool adquiere un rol crucial en configuraciones donde se tienen más threads que CPUs => **programas más rápidos y eficientes.**

Para crear un **EJECUTOR** que **use un pool de threads** se puede invocar a los siguientes métodos de clase de la clase **java.util.concurrent.Executors**:

```
val exec = Executors.newFixedThreadPool(int nThreads)
val exec = Executors.newFixedThreadPool(int nThreads, ThreadFactory threadFact)
```

Creará un pool de **threads** que reusa un conjunto finito de **threads**. En el 2do método se usa el objeto **ThreadFactory** para crear los **threads** necesarios.

```
val exec = Executors.newCachedThreadPool()
val exec = Executors.newCachedThreadPool(ThreadFactory threadFact)
```

Creará un pool de **threads** que crea **threads** nuevos a medida que los necesita y reusa los construidos que están disponibles. El 2do método usa el objeto **ThreadFactory** para crear los nuevos **threads**



# Ejemplo 1

```
import java.util.concurrent.Executors

fun main(args: Array<String>) {

    val executorService = Executors.newCachedThreadPool()

    for (i in 1..10) {

        executorService.execute { . . . }

    }

    executorService.shutdown()

}
```

Tarea Específica

```
val executorService = Executors.newFixedThreadPool(10)
```

# Ejemplo 2

```
import java.util.concurrent.Executors

fun main(args: Array<String>) {

    val executorService = Executors.newSingleThreadExecutor()

    for (i in 1..10) {
        executorService.execute { . . . }
    }

    executorService.shutdown()
}
```

En este ejemplo una tarea se completa en el mismo orden en que es recibida y antes de comenzar una nueva.

Tarea Específica

Un **SingleThreadExecutor** es similar a **FixedThreadPool** con un pool de un único thread. Las tareas se encolan, cada tarea se ejecuta una vez que finaliza la previa. Todas usan el mismo thread. Este ejecutor **serializa** las tareas. Es útil para tareas que necesitan ejecutarse continuamente, por ej: escuchan conexiones entrantes, tareas que actualizan logs remotos o locales o que hacen *dispatching* de eventos. Otro ej: tareas que usan el filesystem evitando manejar la sincronización

# Compartir Recursos

## Condición de Carrera

```
class Counter {  
  
    private var c = 0  
  
    fun increment() {  
        c++  
    }  
  
    fun decrement() {  
        c--  
    }  
  
    fun value(): Int {  
        return c  
    }  
}
```

Si un mismo objeto **Counter** es **referenciado** por múltiples threads (por ejemplo A y B), la interferencia entre estos threads provocaría que el comportamiento de los métodos **increment()** y **decrement()** NO sea el esperado

Recuperar el valor actual de c.  
Incrementarlo/Decrementarlo en 1.  
Guardar en c el nuevo valor

¿Qué pasa si el thread A invoca al increment() al mismo tiempo que el thread B invoca decrement() sobre la misma instancia de Counter?

Si el valor inicial de c es 0, podría ocurrir lo siguiente:

**Thread A:** Recupera c. (c=0)

**Thread B:** Recupera c. (c=0)

**Thread A:** Incrementa el valor recuperado; resultado es c=1.

**Thread B:** Decrementa el valor recuperado; resultado es c=-1 (lo hace antes que A guarde el valor).

**Thread A:** Guarda el resultado en c; c=1

**Thread B:** Guarda el resultado en c; c=-1

**Condición de Carrera**

# Compartir Recursos

- Hasta ahora vimos ejemplos de **threads asincrónicos** que no comparten datos ni necesitan coordinar sus actividades.
- Con multithreading hay situaciones en que dos o más threads intentan acceder a los mismos recursos en el mismo momento. Se debe evitar este tipo de colisión sobre los recursos compartidos (durante períodos críticos): acceder a la misma cuenta bancaria en el mismo momento, imprimir en la misma impresora, etc. Ejemplo de la clase Counter
- Para resolver el problema de colisiones, todos los esquemas de multithreading establecen *un orden para acceder al recurso compartido*. En general se lleva a cabo usando una cláusula que *bloquea* (lock) el código que accede al recurso compartido y así solamente de a un thread a la vez se accede al recurso. Esta cláusula implementa **exclusión mutua**.
- Java provee soporte para **exclusión mutua** mediante la palabra clave **synchronized**.

# Compartir Recursos

- Cada objeto contiene un **lock** único llamado **monitor**. Cuando invocamos a un **método synchronized**, el objeto es “bloqueado” (locked) y ningún otro método **synchronized** sobre el mismo objeto puede ejecutarse hasta que el primer método termine y libere el **lock del objeto**.
- El **lock** del objeto es único y compartido por todos los métodos y **bloques synchronized** del mismo objeto. Este **lock** evita que el recurso común sea modificado por más de thread a la vez.

```
class Recurso {  
    @Synchronized  
    fun f(): Int { }  
    @Synchronized  
    fun g() { }  
}
```

Si el método f() es invocado sobre un objeto Recurso, el método g() no puede ejecutarse sobre el mismo objeto, hasta que f() termine y libere el lock.

- Es posible definir un bloque **synchronized**: **synchronized (unObjeto) {}**
- Un thread puede adquirir el **lock** de un objeto múltiples veces. Esto ocurre si un método invoca a un segundo método **synchronized** sobre el mismo objeto, quién a su vez invoca a otro método **synchronized** sobre el mismo objeto, etc. La JVM mantiene un contador con el número de veces que el objeto fue bloqueado (lock). Cuando el objeto es desbloqueado, el contador toma el valor cero. Cada vez que un thread adquiere el lock sobre el mismo objeto, el contador se incrementa en uno y cada vez que abandona un método **synchronized** el contador se decrementa en uno, hasta que el contador llegue a cero, liberando el lock para que lo usen otros threads. La adquisición del lock múltiples veces sólo es permitida para el thread que lo adquirió en el primer método **synchronized** que invocó.

# Compartir Recursos

## La cláusula synchronized

```
class SynchronizedCounter {  
  
    private var c = 0  
  
    @Synchronized  
    fun increment() {  
        c++  
    }  
  
    @Synchronized  
    fun decrement() {  
        c--  
    }  
  
    @Synchronized  
    fun value(): Int {  
        return c  
    }  
  
}
```

# Ejemplo

## Productor/Consumidor

```
import java.util.concurrent.TimeUnit
import kotlin.concurrent.thread
```

```
fun main(args: Array<String>) {
    val bolsa = Bolsa()
    val i = 1
    thread() {
        Thread.currentThread().name = "${i}"
        for (j in 0..9) {
            bolsa.put(j)
            println("Productor#${Thread.currentThread().name} escribió: ${j}")
            TimeUnit.MILLISECONDS.sleep((1L..100L).random())
        }
    }
    thread() {
        Thread.currentThread().name = "${i}"
        for (k in 1..10) {
            println("Consumidor#${Thread.currentThread().name} leyó: ${bolsa.get()}")
            TimeUnit.MILLISECONDS.sleep((1L..100L).random())
        }
    }
}
```

Objeto Compartido



# Ejemplo

## Productor/Consumidor

Para evitar que el Productor y el Consumidor colisionen sobre el objeto compartido Bolsa, esto es, que intenten guardar y leer simultáneamente dejando al objeto en un estado inconsistente, los métodos `get()` y `put()` se declaran **synchronized**

```
class Bolsa {  
  
    private var contenido = 0  
    private var disponible = false  
  
    @Synchronized  
    fun put(value:Int) {  
        // El objeto bolsa fue bloqueada por el Productor  
        .....  
        //el objeto bolsa fue desbloqueado por el Productor  
    }  
  
    @Synchronized  
    fun get(): Int {  
        return contenido  
        //El objeto bolsa fue bloqueada por el Consumidor  
        .....  
        //el objeto bolsa fue desbloqueada por el Consumidor  
    }  
}
```

Secciones críticas



# Productor/Consumidor

¿Qué sucede si el Productor es más rápido que el Consumidor y genera dos números antes que el consumidor pueda consumir el primero de ellos?

.....

Consumidor #1 leyó: 3

Productor #1 escribió: 4

Productor #1 escribió: 5

Consumidor #1 leyó: 5

← El Consumidor perdió el 4

¿Qué sucede si el Consumidor es más rápido que el Productor y consume dos veces el mismo valor?

.....

Productor #1 escribió: 4

Consumidor #1 leyó: 4

Consumidor #1 leyó: 4

Productor #1 escribió: 5

← El Consumidor obtiene el 4 dos veces

En ambos casos el resultado es erróneo dado que el Consumidor debe leer cada uno de los números producidos por el Productor exactamente una vez.

# Cooperación entre Threads

¿Cómo podemos hacer para que el **Productor** y el **Consumidor** cooperen entre ellos?

El **Productor** debe indicarle al **Consumidor** de una manera sencilla que el valor está listo para ser leído y el **Consumidor** debe tener alguna forma de indicarle al **Productor** que el valor ya fue leído.

Además, si no hay nada para leer, el **Consumidor** debe esperar a que el **Productor** escriba un nuevo valor y, el **Productor** debe esperar a que el **Consumidor** lea antes de escribir un valor nuevo.

Para este propósito la clase **Object** provee los siguientes métodos: **wait()**, **wait(milisegundos)**, **notify()** y **notifyAll()**.

Los métodos **wait()**, **wait(milisegundos)**, **notify()** y **notifyAll()** deben usarse dentro de un método o bloque **synchronized**.

**wait()**  
**wait(milisegundos)**  
**notify()**  
**notifyAll()**

El método **wait()** suspende la ejecución del thread y libera el *lock* del objeto, y así permite que otros métodos **synchronized** sobre el mismo objeto puedan ejecutarse.

El método **notifyAll()** “despierta” a todos los threads esperando (**wait()**), compiten por el *lock* y el que lo obtiene retoma la ejecución. El método **notify()** despierta a un thread.

# Productor/Consumidor

```
class Bolsa {  
    private var contenido = 0  
    private var disponible = false  
  
    @Synchronized  
    fun get(): Int {  
        while (!disponible) {  
            (this as java.lang.Object).wait()  
        }  
        disponible = false  
        (this as java.lang.Object).notifyAll()  
        return contenido  
    }  
  
    @Synchronized  
    fun put(value: Int) {  
        while (disponible) {  
            (this as java.lang.Object).wait()  
        }  
        contenido=value  
        disponible = true  
        (this as java.lang.Object).notifyAll()  
    }  
}
```

Libera el lock (del objeto Bolsa) tomado por el Consumidor, permitiendo que el Productor agregue un dato nuevo en la Bolsa y, luego espera ser notificado por el Productor.

El Consumidor notifica al Productor que ya leyó, dándole la posibilidad de producir un nuevo valor.

Libera el lock (del objeto Bolsa) tomado por el Productor, permitiendo que el Consumidor lea el valor actual antes de producir un nuevo valor.

El Productor notifica al Consumidor cuando agregó un dato nuevo en la Bolsa

# Productor/Consumidor

```
fun main(args: Array<String>) {  
    . . .  
    val bolsa = Bolsa()  
    thread() { . . .  
        bolsa.put(j)  
        println("Productor#{Thread.currentThread().name} escribió: ${j}")  
        . . .  
    }  
    thread() { . . .  
        println("Consumidor#{Thread.currentThread().name} leyó: ${bolsa.get()}")  
        . . .  
    }  
}
```

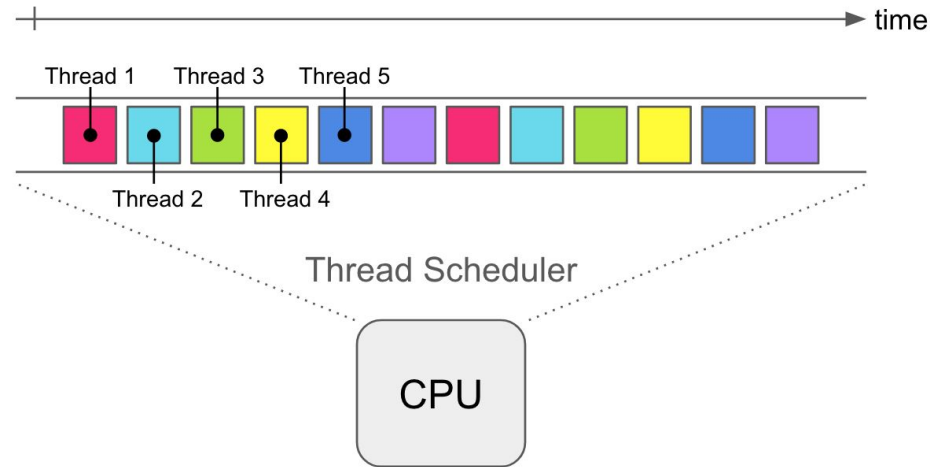
```
Productor #1 escribió: 0  
Consumidor #1 leyó: 0  
Productor #1 escribió: 1  
Consumidor #1 leyó: 1  
Productor #1 escribió: 2  
Consumidor #1 leyó: 2  
Productor #1 escribió: 3  
Consumidor #1 leyó: 3  
.....  
Productor #1 escribió: 9  
Consumidor #1 leyó: 9
```

Salida de la función *main*

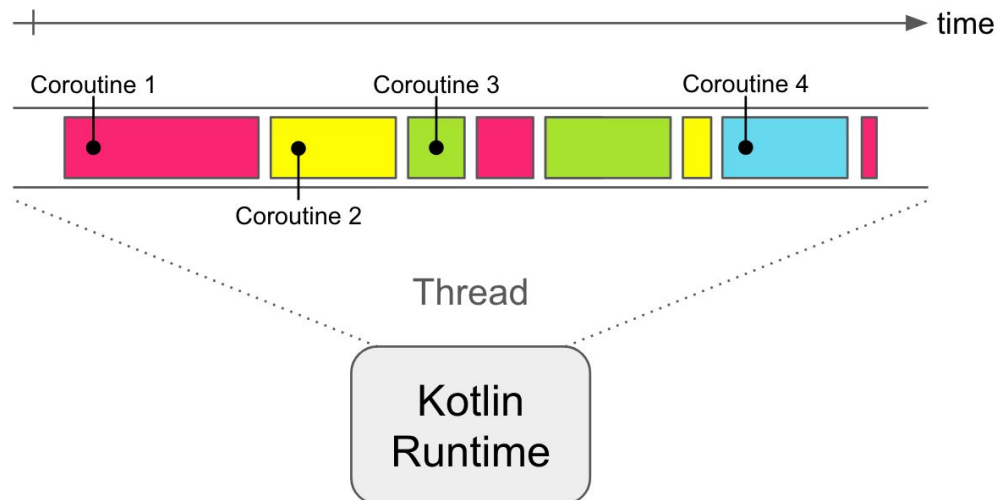
# Concurrencia Corrutinas

# Corrutinas

- Como vimos un **thread** es un flujo de control secuencial dentro de un **proceso**.



- Una **corrutina** es un flujo de control secuencial no bloqueante asíncrono dentro un **thread**, siendo el cómputo más *lightweight* dentro de la programación concurrente.



# Corrutinas vs Threads

- Las **corrutina** proveen concurrencia pero no paralelismo y pueden suspenderse sin bloquear el **thread**.
- Las **corrutinas** son similares a los **threads** en la programación concurrente tradicional, pero están basadas en multitareas cooperativas.
- Los **threads** son siempre globales, mientras que las **corrutinas** tienen alcance.
- El *Scheduling* de las **corrutinas** es *non-preemptive* y es realizado por el lenguaje o el programador. Los cambios entre diferentes contextos de ejecución lo hacen las propias **corrutinas** en lugar del sistema operativo o la máquina virtual.
- Las **corrutinas** se caracterizan por:
  - Ser muy eficientes.
  - No utilizan *context switching*.
  - No reservan espacio extra en la pila de ejecución.
  - No requieren de sincronización.

# Mi primera corrutina en Kotlin

- Las corrutinas forman parte del paquete **kotlinx.coroutines**, por lo que se necesita en el proyecto especificar la siguiente dependencia:  
*implementation("org.jetbrains.kotlin:kotlinx-coroutines-core:1.4.2")*.
- Para iniciar una corrutina se debe usar un constructor (principalmente: **launch**, **runBlocking**, **async**) y pasar las sentencias a ejecutar como un bloque de código.
- Por ejemplo, veamos un programa que imprime *"Hello Word"* con corrutinas.

```
import kotlinx.coroutines.*  
  
fun main() = runBlocking { //inicia una nueva corrutina y bloquea su hilo contenedor  
    launch { // dispara una nueva corrutina y continua.  
        delay(1000L) // non-blocking delay de 1 segundo (unidad default ms)  
        println("World!") // imprime luego del delay  
    }  
    println("Hello") // la corrutina principal continúa mientras que la anterior se suspende  
}
```

Hello  
World!



# Constructor Launch

- El constructor de corrutina **launch** devuelve un **Job** que es el *handle* de la corrutina lanzada.
- El *handle* de una corrutina puede ser usado para esperar explícitamente la finalización o la cancelación la misma.

```
val job = launch { // lanzar una nueva corrutina y
                  //mantiene una referencia a su Job
    delay(1000L)
    println("World!")
}

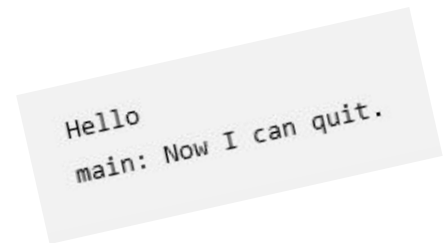
println("Hello")
job.join() // espera hasta que la corrutina se complete
println("Done")
```



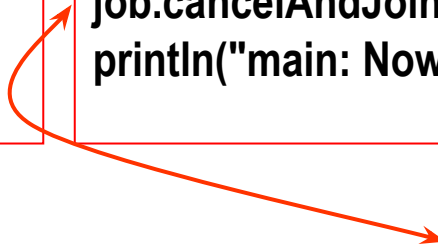
Hello  
World!  
Done

```
val job = launch { // lanzar una nueva corrutina y
                  //mantiene una referencia a su Job
    delay(1000L)
    println("World!")
}

println("Hello")
job.cancelAndJoin() // cancela y espera a que se complete
println("main: Now I can quit.")
```



Hello  
main: Now I can quit.



```
job.cancel() // cancela sin espera
```

# Un ejemplo más sofisticado

- El constructor **launch** puede tomar un parámetro opcional: **CoroutineContext** que especifica un conjunto de elementos como el **dispatcher**, el **CoroutineName**, etc. que determinan el contexto de la corrutina.
- Otro parámetro opcional del **launch** es el **CoroutineStart** que define las opciones de arranque: **DEFAULT**: inicia inmediatamente, **LAZY**: sólo inicia invocando al **start()**, **ATOMIC**: ejecuta atómicamente o **UNDISPATCHED**: ejecuta inmediatamente hasta el primer punto de suspensión en el thread actual.

```
fun main() = runBlocking {  
    val jobs: List<Job> = List(10) { // lista de 10 jobs  
        launch(  
            { Dispatchers.Default +  
              CoroutineName("#$it")  
              CoroutineStart.LAZY } // no se inicia al instante  
            {  
                delay(1000L)  
                println("Hello World from $it!")  
            }  
        )  
    }  
    jobs.forEach { it.start() } //inicia las 10 corrutinas simultaneamente  
}
```

Use un pool threads compartidos en background.

Contexto de la corrutina

Nombre especificado explícitamente

Define la opción de inicio

Hello World from 1!  
Hello World from 2!  
Hello World from 3!  
Hello World from 4!  
Hello World from 5!  
Hello World from 6!  
Hello World from 7!  
Hello World from 8!  
Hello World from 9!  
Hello World from 0!

# Dispatchers y threads

- El **dispatcher** de la corrutina que determina qué thread o threads se utilizaran para la ejecución, pudiendo confinar la ejecución a un thread específico, despacharla a un pool de threads o dejar que se ejecute sin confinar.
- Alternativas para el uso de **dispatchers**:
  - Usar uno de un grupo de varias implementaciones de **CoroutineDispatcher**:
    - **Dispatchers.Default**: Usa un pool threads compartidos en background.
    - **Dispatchers.Main**: Se limita al Main thread y por lo general es single-threaded .
    - **Dispatchers.IO**: Está diseñado para transferir tareas IO bloqueantes a un grupo compartido de threads.
    - **Dispatchers.Unconfined**: No se limita a ningún thread específico, permite que la corrutina se reanude en cualquier subproceso utilizado por la función de suspensión correspondiente, sin imponer ninguna política de subprocesos específica.
  - Crear un único thread dedicado para ejecutar la corrutina con **newSingleThreadContext**, pero resultando muy caro desde el punto de vista del uso de recursos.
  - Usar **java.util.concurrent.Executor** arbitrario que pueda convertirse en un **dispatcher** con la función de extensión **asCoroutineDispatcher**.

# Un ejemplo usando distintos Dispatchers

```
import kotlinx.coroutines.*
import java.util.concurrent.*
```

```
fun main() = runBlocking<Unit> {
```

```
    launch { // contexto de la corrutina padre, principal runBlocking
        println("Hello World in the main runBlocking and working in thread ${Thread.currentThread().name}")
    }
```

```
    launch(Dispatchers.Default) { // usa el dispatcher Dispatchers.Default
        println("Hello World using Dispatchers.Default and working in thread ${Thread.currentThread().name}")
    }
```

```
    launch(Dispatchers.IO) { // usa el dispatcher Dispatchers.IO
        println("Hello World using Dispatchers.IO and working in thread ${Thread.currentThread().name}")
    }
```

```
    launch(Dispatchers.Unconfined) { // usa el dispatcher Dispatchers.Unconfined
        println("Hello World using Dispatchers.Unconfined and working in thread ${Thread.currentThread().name}")
    }
```

```
    launch(newSingleThreadContext("MyOwnThread")) { // crea su propio thread
        println("Hello World using a newSingleThreadContext and working in thread ${Thread.currentThread().name}")
    }
```

```
    launch(Executors.newSingleThreadExecutor().asCoroutineDispatcher()) { // usa un java.util.concurrent.Executor, newSingleThreadExecutor
        println("Hello World using a java executor (newSingleThreadExecutor) and working in thread ${Thread.currentThread().name}")
    }
```

Hello World using Dispatchers.Default and working in thread DefaultDispatcher-worker-  
Hello World using Dispatchers.IO and working in thread DefaultDispatcher-worker-  
Hello World using Dispatchers.Unconfined and working in thread main @coroutine#5  
Hello World using a java executor (newSingleThreadExecutor) and working in thread  
Hello World in the main runBlocking and working in thread main @coroutine#2  
Hello World using a newSingleThreadContext and working in thread MyOwnThread @c

# Modificador suspend

- En Kotlin, las corrutinas se usan para implementar **funciones de suspensión** y pueden cambiar contextos sólo en los **puntos de suspensión**.
- El modificador **suspend** permite lanzar un método en una corrutina.
- Los métodos marcados con **suspend** sólo pueden ser invocados desde una corrutina o desde otro método **suspend**.


```
import kotlinx.coroutines.*  
fun main() = runBlocking {  
    launch { doWorld() }  
    println("Hello")  
}  
  
// función suspend  
suspend fun doWorld() {  
    delay(1000L)  
    println("World!")  
}
```



# Alcance de Corrutinas

- Utilizando el constructor **coroutineScope** se crea un ámbito de corrutina que no finaliza hasta que todos los hijos lanzados hayan terminado.
- Para crear corrutinas de nivel superior, alcance de aplicación, se utiliza el constructor **GlobalScope.launch**.
- Los constructores **runBlocking** y **coroutineScope** difieren principalmente en que el primero bloquea el thread actual, mientras que el segundo sólo suspende; liberando el computo para otros usos.

```
fun main() = runBlocking {  
    doWorld()  
}  
  
suspend fun doWorld() = coroutineScope {  
    launch {  
        delay(1000L)  
        println("World!")  
    }  
    println("Hello")  
}
```




*// Ejecuta secuencialmente doWorld seguido de "Done"*

```
fun main() = runBlocking {  
    doWorld()  
    println("Done")  
}
```

*// Ejecuta concurrentemente ambas secciones*

```
suspend fun doWorld() = coroutineScope {  
    launch {  
        delay(2000L)  
        println("World 2")  
    }  
    launch {  
        delay(1000L)  
        println("World 1")  
    }  
    println("Hello")  
}
```



**coroutineScope** se utiliza para múltiples operaciones concurrentes

Hello  
World 1  
World 2  
Done

# Constructor async

- El constructor de corrutina **async** devuelve un objeto **Deferred<T>**.
- Es necesario invocar a **await()** para obtener el objeto resultado de tipo T del **Deferred<T>**.
- Una función común no puede llamar al **await**, se debe usar **launch** para lanzar una corrutina nueva desde una función.
- Se debe usar el **async** solo cuando se esté dentro de una **corrutina** o de una función **suspend**.

```
// Ejecuta secuencialmente doWorld seguido de "Done"
fun main() = runBlocking {
    doWorld()
    println("Done")
}

// Ejecuta concurrentemente ambas secciones.
suspend fun doWorld() = coroutineScope {
    val res1 = async {
        delay(2000L)
        "World 2"
    }
    val res2 = async {
        delay(1000L)
        "World 1"
    }
    println("Hello ${res1.await()}, ${res2.await()}")
}
```

Hello World 2, World 1  
Done

# Flujo asíncrono flow

- **Flow<T>** representa un flujo de datos asíncrono que emite valores secuencialmente.
- Usualmente representan *cold streams*, no se produce ningún valor si no hay nadie que lo colecte.
- Las principales maneras de construir un flujo son:
  - **flowOf()** crea un flujo a partir de un conjunto fijo de valores.
  - **asFlow()** construye un flujo sobre distintos conjuntos de elementos.
  - **flow {}** fabrica un flujo a partir de llamadas secuenciales a la función **emit**.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
```

```
fun simple(): Flow<Int> = flow { // constructor del flow
    for (i in 1..3) {
        delay(100)
        emit(i) // emite el proximo valor
    }
}
```

```
fun main() = runBlocking{
    launch {
        for (k in 1..3) {
            println("Hello Wold_$k")
            delay(100)
        }
    }
    simple().collect { value -> println(value) } // Recibe los valores del flujo
}
```



Hello Wold\_1  
1  
Hello Wold\_2  
2  
Hello Wold\_3  
3

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
```

```
fun main() = runBlocking {
    val sum = (1..5).asFlow()
        .map { it * it }
        .reduce { a, b -> a + b }
    println(sum)
}
```

55

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
```

```
fun main() = runBlocking {
    val sum = flowOf(1, 2, 3, 4, 5)
        .map { it * it }
        .reduce { a, b -> a + b }
    println(sum)
}
```

55




# Canales channel y channelFlow

- Un **canal** es conceptualmente muy similar a cola bloqueante con dos operaciones en suspensión **send** y **receive**.
- **channelFlow** es utilizado para crear flujos destinados a trabajar de forma concurrente y en lugar de utilizar la función **emit** utiliza **send**.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
```

```
fun main() = runBlocking {

    val channel = Channel<Int>()
    launch {
        for (x in 1..5) channel.send(x * x)
        channel.close() // cierra el canal
    }
    launch {
        for (message in channel) println("Hello World_$message")
    }
    println("Done!")
}
```



```
Done!
Hello World_1
Hello World_4
Hello World_9
Hello World_16
Hello World_25
```

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
```

```
fun canalFlow(): Flow<Int> = channelFlow {
    launch {for (i in 1..2) send(i) }
    launch {for (i in 3..5) send(i) }
}
```

```
fun main() = runBlocking{
    launch {
        for (k in 1..3) {
            println("Hello Wold_$k")
            delay(100)
        }
    }
    canalFlow().collect{value -> println(value)}
}
```



```
Hello Wold_1
1
2
3
4
5
Hello Wold_2
Hello Wold_3
```

# Canales vs Flujos

- La primera y más obvia diferencia es que **los canales empiezan a emitir datos inmediatamente** sin importar si algún consumidor recibe los datos.
- Los **flujos** son una buena opción para el **procesamiento secuencial de datos**.
- Los canales se deben cerrar explícitamente una vez finalizado su objetivo para evitar pérdidas.
- Para delimitar las emisiones al ciclo de vida de un determinado componente se recomienda el uso de los flujos y si por el contrario la generación de datos no debe estar sujeta a un ciclo de vida específico, se debe usar canales.