1. Implemente en Kotlin una aplicación que muestre la hora actual en consola y la actualice cada 1 segundo. Evalúe distintos mecanismos para hacerlo.

```
import java.time.LocalDateTime
import java.util.concurrent.TimeUnit

fun main(args: Array<String>) {
   for (i in 1..10){
       println("son las:" + LocalDateTime.now())
       TimeUnit.SECONDS.sleep(1)
   }
}
```

la básica

```
fun main(args: Array<String>) {
   val t= thread {
       for (i in 1..10){
           println("son las:" + LocalDateTime.now())
           TimeUnit.SECONDS.sleep(1)
       }
   }
t.join()
}
```

Crea yun hilo pero medio al dope porque tenés el hilo de la ejecución principal y este y no hay trabajo para hacer. Bloquea al hilo main mientras se ejecuta el código t.join() se usa para esperar a que termine el hilo.

```
fun main() = runBlocking {
   repeat(10) {
       println("Hora: ${LocalDateTime.now()}")
       delay(1000)
   }
}
```

No bloquea hilos reales, es más eficiente y estilo moderno Kotlin

```
class TestSynchronized(id: String) : Thread(id) {
    var frase: Array<String> = arrayOf("UNLP", "PÚBLICA", "AHORA", "Y", "SIEMPRE")
    override fun run() {
        synchronized(System.out) {
            for (palabra in frase)
                println("${this.name} : ${palabra} ")
        }
    }


}
fun main(args: Array<String>) {
    val t1 = TestSynchronized("Thread 1")
    val t2 = TestSynchronized("Thread 2")
    val t3 = TestSynchronized("Thread 3")
    t1.start()
    t2.start()
    t3.start()
}
```

a.- ¿Cuál es el efecto del synchronized(System.out)?
Synchronized se usa para exclusión mutua. En este caso entiendo que lo que hace es que asegura que solo uno a la vez está imprimiendo.
Este bloque sincronizado asegura que la operación de impresión sea atómica respecto a los demás threads. Así se evita que dos threads escriban en `System.out` al mismo tiempo y produzcan una salida intercalada o corrupta.
Si no estuviera, podría pasar algo como
Thread 1: UNLP
Thread 2: UNLP
Thread 1: PÚBLICA
Thread 3: UNLP
Thread 2: PÚBLICA
Thread 1: AHORA

b.- ¿Qué tipo de lock hace el código dado?
El código está usando un lock implícito (monitor) sobre el objeto System.out. Es un mecanismo de sincronización de tipo mutex, donde System.out actúa como monitor.

3. Implemente una aplicación que simule una carrera de 100 metros, donde cada participante está representado por un objeto thread. Para ello, cree un programa que muestre por consola la cantidad de metros recorrida por cada corredor.
a.- Use un ejecutor con un pool de tamaño 5 para ejecutar. Luego cambie el tamaño del pool a 3 y observe la ejecución de los threads.
Si ponés 5 → corren 5 corredores a la vez
Si ponés 3 → corren solo 3 a la vez, los otros esperan.

b.- Supongamos que se quiere saber si un corredor abandona la carrera, retornando algún valor predefinido o en el peor de los casos, disparando una excepción. Analice la interface Callable, usando la documentación de la API y observe sus ventajas.

Listo, vamos a bajarlo a tierra 🤝

Tu código original mezclaba 2 ideas distintas: **Callable** y **execute**.

Para hacerlo simple, olvidate de pools, futuros y toda la vuelta complicada por ahora.

✅ **¿Qué cambia de (a) a (b)?**

Parte	(a) Runnable	(b) Callable
¿Devuelve algo el hilo?	❌ No	✅ Sí (`String`)
¿Puede tirar excepción?	❌ No checked	✅ Sí
¿Se ejecuta con `execute`?	✅ Sí	❌ No
¿Se usa `submit()` y `Future`?	❌ No	✅ Sí

Clave: Si usás `Callable`, NO llamás `call()` vos, sino que el Executor lo llama por vos con `submit()`.