Alerta de documento super denso.

1 - Programação Assíncrona – Estilo e técnicas de programação que considera “tarefas acontecendo sem precisar bloquear outras” – várias situações exigem esse tipo de consideração. (Um exemplo feito em draw.io, em dois fluxogramas: fazendo café).

1.1 – Threads (JAVA-LANG) – Classe em java que permite a ‘criação manual’ de uma Thread para que seja executada uma tarefa. Essa classe, por ser de baixo nível, joga a execução diretamente ao sistema operacional (android, no caso) e ele por sua vez, delega a um thread do processador do celular.

1.1.1 - Métodos da sintaxe Thread (funções)

1.1.1.1 – método start – função java – serve para iniciar o código definido num thread diferente.

1.1.1.2 – método sleep – função java – serve para fazer uma pausa especificada em milissegundos.

1.1.1.3 – método join – função java que faz a thread atual aguardar até que outra thread termine. Apesar do nome, não confundir com o Join que vem aí, na frente.

1.1.1.4 - método interrupt – função java – serve para sinalizar que a thread deve ser interrompida.

1.1.1.5 - método run – função java sobrescrevível e que pode rodar de maneira síncrona (sem ‘trocar de thread’), em contraponto: método start() por padrão troca de thread para rodar.

1.2 - Funções assíncronas – Aqui são as funções que por via de regra não são bloqueantes, com observações a seguir:

1.2.1 - Suspend fun (sem exemplos) – O coração da execução assíncrona (sem bloqueio) no Kotlin.

1.2.2 – Callback (sem exemplos aqui) – Usamos bastantes call-backs, mas para que ela seja uma assíncrona mesmo, precisa de uma aplicação relativamente específica (veremos a seguir em breve).

1.3 – Corrotina – Abstração de programação assíncrona que permite realizar tarefas concorrentes (paralelas ou não), sem necessariamente bloquear a thread (ou contexto) atual.

1.3.1 – Contexto – É um ambiente definido no código, que faz a gestão de principalmente duas coisas: o pool de threads (threads que são geridas pela própria biblioteca do Kotlin (coroutines)), e ela (automaticamente) define quais threads (ou pool) do sistema operacional (ou processador) vai executar suas tarefas definidas. Pense no contexto como "a sala de operações" onde as tarefas acontecem. O Dispatcher é "quem faz o trabalho" (os trabalhadores), e o Job é "o gerente" que coordena o trabalho.

1.3.1.1 – Dispatchers – é QUEM e ONDE vão ser feitas as operações. Cada pool de threads é especializado em fazer uma coisa. O Dispatcher aponta para a biblioteca qual o dispatcher mais adequado naquele caso, e é o programador que vai delegar ao thread especifico a atividade a ser executada.

1.3.1.1.1 – Dispatchers.Main – Importante para operações na thread principal, como atualizações de UI por exemplo.

1.3.1.1.2 – Dispatchers.IO – Ideal para leitura e gravação de dados, não importando tanto onde (busca em API é uma leitura de dados, ou leitura/gravação de dados do room database...) – PS: IO é de In/out – entrada e saída.

1.3.1.1.3 – Dispatchers.Default – Esse é utilizado para operações matemáticas robustas, operações e cálculos pesados.

1.3.1.1.4 – Dispatchers.Unconfined – Não vincula uma atividade a uma thread específica (ou seja, o pool de threads dele é difuso), e por isso não teria problemas utilizá-lo como dispatcher parâmetro em testes leves. Em testes não oferece ferramentas de controle de tempo, por isso aconselhamos a usar sempre em casos bem básicos onde manipulações de tempo e agendamentos (escalonamentos) não são necessários. Dica: use apenas em casos específicos, tipo em testes leves.

1.3.1.1.5 – Dispatcher personalizado – Você quem cria esse cara, o Kotlin permite que você crie um personalizado – pergunte ao GPT como fazer isso, vou evitar colocar aqui para não ficar muito extenso, mas saiba que isso é possível!

1.3.1.1.6.0 – UnconfinedTestDispatcher – Tipo especial de dispatcher para ambiente de testes (kotlinx-coroutines-test) – Usa um agendador (escalonador) padrão do kotlin para testes, sem controle de tempo de maneira explicita. (no construtor dessa classe, você pode colocar o TestCoroutineScheduler para que o tempo possa ser controlado de maneira mais explícita).

1.3.1.1.6.1 – TestCoroutineScheduler – É uma classe de importância média, que extende outra classe de contexto no kotlin. É basicamente um agendador (escalonador) aberto, e ao usá-lo você pode controlar um pouco mais detalhadamente a questão dos tempos na hora de fazer seus testes (caso seja necessário). O controle de tempo é feito de maneira virtual.

1.3.1.1.6.1.1 – Métodos de uso do agendador – O Agendador é uma ferramenta robusta no que diz respeito a controlar os tempos e momentos de execução das atividades, aqui estão seus métodos

1.3.1.1.6.1.2 – advanceTimeBy(milliseconds: Long) – Avança o tempo virtual em uma quantidade específica de milissegundos. Serve pra simular o avanço do tempo em testes que dependem de delay ou outros eventos baseados em tempo.

1.3.1.1.6.1.3 – advanceUntilIdle() – Avança o tempo até que todas as tarefas agendadas estejam concluídas ou não restem tarefas pendentes. Usar para garantir que todas as tarefas agendadas no fluxo estejam concluídas sem precisar especificar o tempo.

1.3.1.1.6.1.4 – currentTime – Retorna o tempo virtual atual em ms. Serve para verificar em que momento do tempo virtual a execução está ocorrendo.

1.3.1.1.6.1.5 – runCurrent() – Executa imediatamente todas as tarefas agendadas para o momento atual do tempo virtual. Serve para forçar a execução de tarefas que estão prontas para serem processadas no tempo atual.

1.3.1.1.6.1.6 – hasPendingTasks() – Retorna um booleano indicando se há tarefas para serem executadas. Serve para verificar se há algo agendado que ainda não foi processado.

1.3.1.1.6.1.7 – pauseDispatcher() – Pausa o dispatchers associado ao scheduler, impedindo que novas tarefas sejam executadas automaticamente. Serve para controlar manualmente quando as tarefas podem ser executadas.

1.3.1.1.6.1.8 – resumeDispatcher() – Retoma a execução de tarefas no dispatchers associado ao scheduler. Serve para liberar as tarefas que estavam pausadas.

1.3.1.2 – Job – Não tão importante para nossos objetivos, portanto ficará de lado nessa apresentação.

1.3.2 – Escopos – São os gestores gerais da execução e do ciclo de vida das corrotinas como um todo.

1.3.2.1 – Escopos baseados em classes (“aplicação maior”)

1.3.2.1.1 – CoroutineScope – interface **importantíssima** para a criação e gestão de contextos (1.3.1)

1.3.2.1.2 – GlobalScope – É uma extensão da CoroutineScope. Ele cria um escopo que não é vinculado a nenhum ciclo de vida específico. Corrotinas iniciadas nele só serão canceladas quando o programa terminar. Evite usar em produção, pois pode causar o vazamento de corrotinas.

1.3.2.1.3 – backgroundScope – Extensão da CoroutineScope – é um escopo de fundo que serve para lidar com operações o quão extensas forem necessárias e, no ambiente de testes nos é útil na dinâmica de troca de contextos que você fez no seu código original, no que está sendo testado agora, mas especificamente na coleta de estados de fluxos nessas trocas de contextos, e acaba sendo bem adequada por não conhecermos o tempo total da coleta de estados para os estados de fluxo...

1.3.2.1.4 – viewModelScope – Extensão da CoroutineScope – é um escopo de fundo também, mas diferenciando do anterior, ele tem a função em si mesmo, de lidar com o ciclo de vida do próprio viewmodel no android. Corrotinas são canceladas se o viewModel for destruído.

1.3.2.2 – Escopos baseados em funções (“aplicação menor”) – São utilizados para criar escopos locais e temporários dentro de corrotinas – são funções suspensas.

1.3.2.2.1 – coroutineScope – Função suspensa, e todas as atividades dentro dele têm o mesmo JOB. Se uma tarefa falhar, o coroutineScope inteiro lança uma excessão e cancela as outras tarefas/corrotinas.

1.3.2.2.2 – supervisorScope – Mesma coisa do de cima, mas se uma tarefa falhar, ele não lança excessão.

1.3.3 - Métodos de uso

1.3.3.1 - Métodos Utilitários – Controles de tempo e contexto.

1.3.3.1.1 – withTimeout – Função suspensa - cancela a execução do código dentro dela, se o tempo limite for atingido (lança excessão/erro)

1.3.3.1.2 – withTimeoutOrNull - Função suspensa – semelhante a withtimeout mas ela retorna null, não excessão/erro.

1.3.3.1.3 – withContext – Função suspensa – serve para trocar o CONTEXTO (em geral, o pool de threads, mas também o job se necessário).

1.3.3.1.4 – Join – Função suspensa – Faz a corroutine atual aguardar a conclusão de outra corroutine.

1.3.3.1.5 – await – Função suspensa – Aguarda o resultado de uma corroutine criada com async (1.3.3.2.1)

1.3.3.1.6 – delay – Função suspensa – Suspende a execução da corroutine por um tempo não bloqueando a thread.

1.3.3.1.7 – yield – Função suspensa – Cede o contexto para outras corrotinas, mas não força a alternância imediata. Ela permite que o contexto seja liberado (caso necessário, mas não obrigatoriamente).

1.3.3.2 – Métodos de Execução – Controle de início, “espera” e bloqueio de thread.

1.3.3.2.1 – Async – Função especial, extensora da interface CoroutineScope – Método de iniciar uma corrotina num contexto específico, mas async implica o retorno de um valor (o await (1.3.3.1.5) serve para coletar esse valor quando ele for emitido).

1.3.3.2.2 – Launch – Função especial, extensora da interface CoroutineScope – Método de iniciar uma corrotina num contexto específico, sem retornar valor.

1.3.3.2.3 – runBlocking – Função especial: não é suspensa, mas funciona como corrotina (feita com programação de alto nível) – Ela serve para bloquear a thread atual até que todas as corrotinas dentro dela dela terminem. Não é recomendado utilizar em escala, nem em produção. Somente em situações “pequenas” (pouco código de rápida execução), para testes em ambiente didático.

Conceito	Função	Diferença principal
coroutineScope	Cria escopo local e cancela tudo se uma tarefa falhar	Todas as tarefas compartilham o mesmo Job
supervisorScope	Cria escopo local e não cancela outras tarefas se uma falhar	Cada tarefa tem seu próprio Job
withTimeout	Cancela o bloco ao atingir o tempo limite	Lança excessão em caso de timeout
withTimeoutOrNull	Cancela o bloco ao atingir o tempo limite e retorna null	Não lança excessão em caso de timeout
launch	Inicia corrotina sem retorno de valor	Executa tarefas assíncronas sem resultado direto
Async	Inicia corrotina que retorna um valor (via Deferred)	Use await para capturar o resultado