***Structured concurrency***

یک دیزاین پترن است که به ساده سازی و افزایش قابلیت اطمینان در اجرای فرایند های concurrent کمک می‌کند. این دیزاین پترن اطمینان حاصل می‌کند تا تمام فرایند‌های concurrent به صورت ساختار یافته و مدیریت شده اجرا شوند تا با یکدیگر به تداخل بر نخورند و side effect ای بر جای نگذارند.

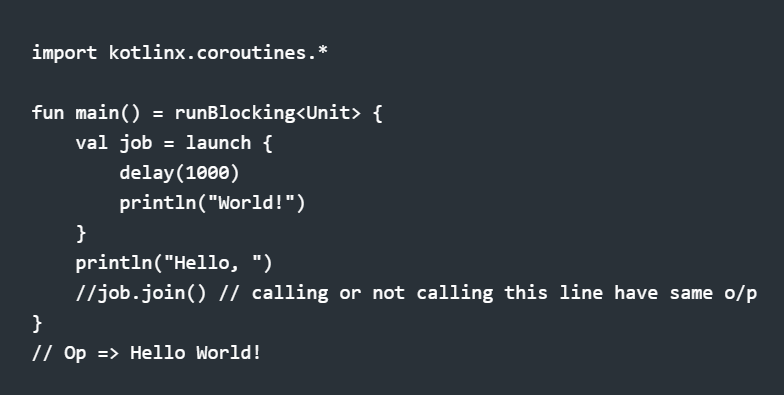
این هدف به کمک محدود کردن اجرای فرایند ها در یک سلسله مراتب خاص (هر فرایند درون یک parent context انجام می‌شود و تضمین می‌شود تا قبل از آن که parent اش خاتمه یابد فعالیتش را تمام کند) محقق می‌شود.

Structured concurrency در کاتلین بر اساس یک رابطه‌ی سلسله مراتبی بین کروتین هاست، هر کروتین یک parent دارد. موقعی که یک کروتین جدید start زده می‌شود این کروتین به child یک کروتین دیگر تبدیل می‌شود.

این رابطه‌ی سلسله مراتبی باقی است تا زمانی که طول عمر تمام کروتین ها به پایان برسد. لازم به ذکر است هنگامی که یک کروتین parent کنسل می‌شود تمام child های آن نیز به صورت سلسله مراتبی از پایین ترین قسمت کنسل می‌شوند این کمک می‌کند تا مطمئن شویم هیچ leak و یا کروتین معلقی وجود ندارد.

یکی از مزایای مهم Structured concurrency آن است که به کمک محدود کردن اجرای کروتین ها به یک context سلسله مراتبی خاص مطمئن می‌شود تا همه‌ی فرایند‎ها مدیریت شده و ریسورس ها در یک زمان قابل پیشبینی آزاد شوند. به کمک این مورد می‌توان بر بسیاری از مشکلات فرایند‌های concurrent مانند race condition و deadlock و resource leak غلبه کرد.

به مثال زیر توجه کنید:



جاب در یک کروتین دیگر به نام runBlocking قرار گرفته و خود runBlocking به فانکشن main محدود شده است. در این مثال ابتدا باید job به اتمام برسد تا runBlocking بسته شود و خود runBlocking باید کامل شود تا تابع main بتواند return کند و بسته شود.

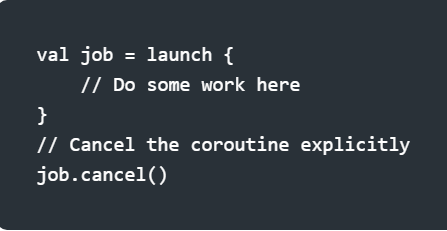
این باعث می‌شود تا مطمئن شویم هنگامی که تابع main بسته می‌شود job و runBlocking به اتمام رسیده است.

***Coroutine Cancellation***

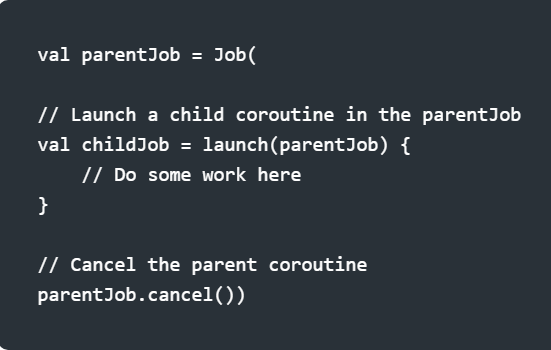
Cancellation فرایند توقف اجرای یک کروتین است قبل از آن که آن کروتین وظیفه‌ی خود را به پایان برساند. موقعی که یک کروتین کنسل می‌شود اجرایش را از نزدیک ترین cancellation point به پایان می‌رساند. یک cancellation point جایی است که کروتین در آن نقطه به دنبال signal توقف می گردد تا در صورتی که آن را پیدا کرد به اجرا خاتمه دهد(به طور مثال فانکشن delay می‌تواند نقش یک cancellation point را بازی کند)

یک کروتین به دو روش می‌تواند کنسل شود:

* **Explicitly: به کمک فانکشن cancel() می‌توان یک کروتین را به صورت explicit متوقف کرد. فانکشن cancel() یک عضو از اینترفیس job است و می‌تواند کروتین را بدون در نظر گرفتن state فعلی آن و به صورت فوری متوقف کند.**



**Automatically: یک کروتین هنگامی که parent اش متوقف می‌شود می‌تواند به صورت اتوماتیک متوقف شود.**



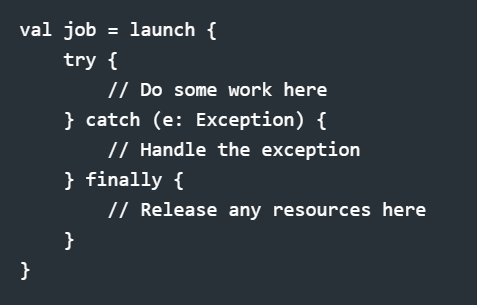
موقعی که یک کروتین کنسل می‌شود، استیت job آن به cancelling تغییر می‌کند و فرزندانش به صورتrecursive  از پایین ترین قسمت سلسله مراتب شروع به کنسل شدن می‌کنند. در طی فرایند cancelling هر کروتین فرصت این را دارد تا ریسورس ‌های خودش را clean up کند.

توجه شود که وقتی یک کروتین کنسل می‌شود الزاما به این معنا نیست که بلاک آن می‌تواند به طور کامل انجام شود. تنها زمانی می‌توان از اجرای کامل بلاک مطمئن شد که کروتین به صورت نرمال complete شده باشد. (پس چگونه می‌توان فرصت آزاد سازی منابع را به دست آورد؟)

***How to handle the Coroutine Cancellation?***

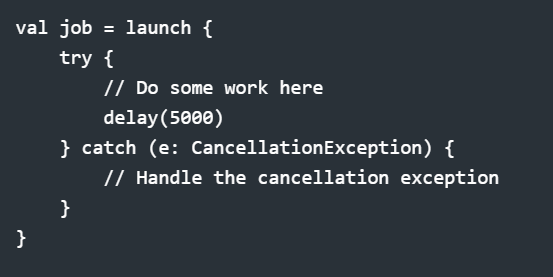
هنگامی که یک کروتین کنسل می‌شود، این نکته حائز اهمیت است که cancellation را به نحوی مدیریت کنیم تا از هر گونه resource leak و خطا‌های دیگر جلوگیری شود. چندین راه برای این امر وجود دارد.

**try-catch-finally: این بلاک می‌تواند برای مدیریت cancellation استفاده شود. بلاک finally می‌تواند برای آزاد سازی ریسورس‌ها استفاده شود زیرا حتی اگر کروتین متوقف شده باشد finally همچنان اجرا می‌شود.**



**withContext(NonCancellable): این فانکشن می‌تواند برای اجرای یک بلاک از کد‌هایی استفاده شود که در coroutine context فعلی قابل کنسل نیستند.**

**kotlinx.coroutines.delay(): این فانکشن یک فانکشن cancellable است و هنگامی که سیگنال cancel را دریافت می‌کند CancellationException ای را throws می‌کند. ما با دریافت این exception می‌توانیم فرصت آزاد سازی resource ها را به دست آوریم.**



......