***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.

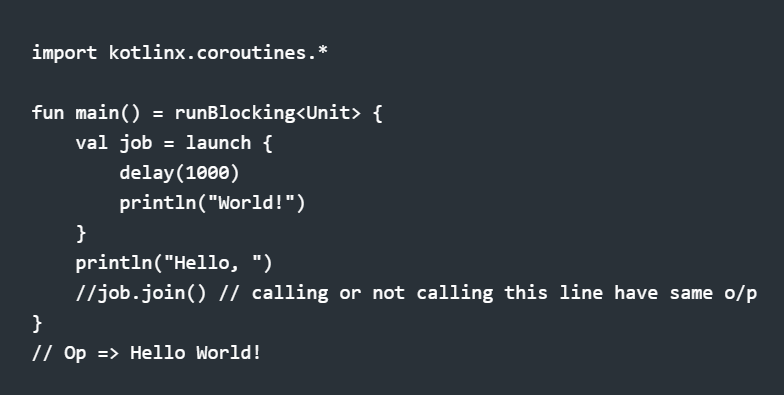
این هدف به کمک محدود کردن اجرای فرایند ها در یک سلسله مراتب خاص (هر فرایند درون یک parent context انجام می‌شود و تضمین می‌شود تا قبل از آن که parent اش خاتمه یابد فعالیتش را تمام کند) محقق می‌شود.

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.

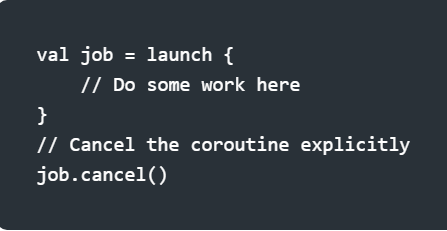
این باعث می‌شود تا مطمئن شویم هنگامی که تابع main بسته می‌شود job و runBlocking به اتمام رسیده است.

***Coroutine Cancellation***

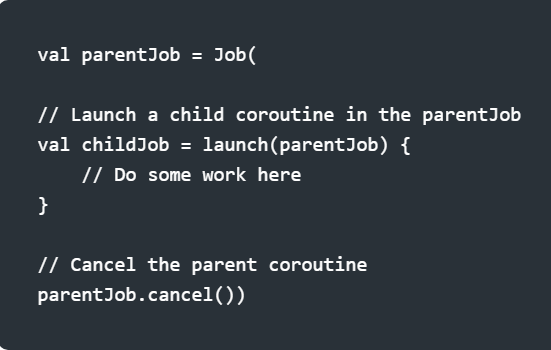
Cancellation فرایند توقف اجرای یک کروتین است قبل از آن که آن کروتین وظیفه‌ی خود را به پایان برساند. موقعی که یک کروتین کنسل می‌شود اجرایش را از نزدیک ترین cancellation point به پایان می‌رساند. یک cancellation point جایی است که کروتین در آن نقطه به دنبال signal توقف می گردد تا در صورتی که آن را پیدا کرد 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.**



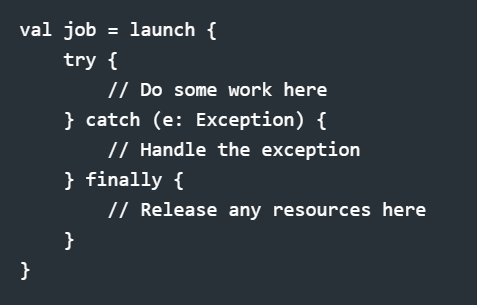
موقعی که یک کروتین کنسل می‌شود، استیت job آن به cancelling تغییر می‌کند و فرزندانش به صورتrecursive  از پایین ترین قسمت سلسله مراتب شروع به کنسل شدن می‌کنند. در طی فرایند cancelling هر کروتین فرصت این را دارد تا ریسورس ‌های خودش را clean up کند.

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؟)

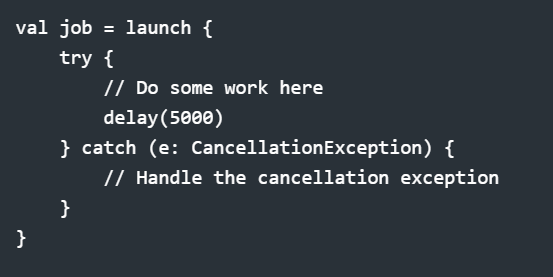
***How to handle the Coroutine Cancellation?***

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 سیگنال کنسل را دریافت کرده 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.**



***How to ensure that finally block is also executed even when coroutine is cancelled?***

**در کاتلین کروتین، یک بلاک finally برای اجرای کد‌هایی به کار می‌رود که باید صرف نظر از آن که exception ای اتفاق افتاده است یا خیر اجرا شوند. در مواردی که یک کروتین سیگنال کنسل را دریافت کرده، مهم است تا مطمئن شویم که بلاک finally به طور کامل اجرا می‌شود یا خیر**

**هنگامی که یک کروتین کنسل می‌شود، اجرای آن متوقف می‌شود و یک cancellation exception به سمت والد آن پرتاب می‌شود، این exception در سلسله مراتب آن قدر بالا می‌رود تا به root برسد.**

**هنگامی که یک کروتین کنسل می‌شود، بلاک finally قبل از آن که کروتین به صورت حقیقی متوقف شود (رسیدن exception به root در سلسله مراتب) اجرا می‌شود. این بلاک برای اعمالی مانند آزاد سازی منابع، بستن ارتباط شبکه یا دیتابیس و ... کاربرد دارد. اما گاهی از اوقات یک یا برخی از این اعمال ممکن است خودشان cancellable و زمانبر باشند در چنین شرایطی وقوع exception در عملکرد آن‌ها می‌تواند سیستم را در شرایط ناپایداری قرار دهد.**

**برای آن که مطمئن شویم بلاک finally هنگام کنسل شدن کروتین به طور کامل اجرا می‎شود چند راه حل وجود دارد:**

* کال کردن yield() در بلاک finally: هنگامی که تابع yield() کال می‌‎شود کروتین را suspend می‌کند و منتظر می‌شود تا مجدد resume شود. استفاده از yield منجر می‌شود تا مطمئن شویم بلاک finally در هنگام کنسل شدن کروتین حتما اجرا می‌شود. اما چگونه؟

هنگامی که تابع yield() کال می‌‎شود کروتین را suspend می‌کند و منتظر می‌شود تا مجدد resume شود. هنگامی که یک کروتین کنسل می‌شود نمی‌تواند مجددا resume شود، اما کال کردن yield در finally منجر می‌شود تا کروتین مجددا در صف ready coroutine ها قرار بگیرد به این ترتیب finally فرصت پیدا میکند تا قبل از پایان حقیقی کروتین کارش را به پایان برساند.