



Exception Handling

1

Cancellation and Exception Handling

- Cancellation is important for avoiding doing more work than needed which can waste memory and battery life.
- Proper exception handling is key to a great user experience.

2

Exception Handling

- Exception and error handling is an integral part of asynchronous programming.
- It's important to know *how errors and exceptions are propagated* through the process.

3

CoroutineScope

- To start & control the **lifecycle** of coroutines, they should be created in `CoroutineScope`.
- A `CoroutineScope` keeps track of any coroutine created using `launch` or `async`.
- Coroutines can be canceled by calling `scope.cancel()` at any time.
- In Android, we have `viewModelScope` and `lifecycleScope`

```
val scope = CoroutineScope(CoroutineContext(Job() + Dispatchers.Main))
val job = scope.launch {
    // new coroutine
}
```

4

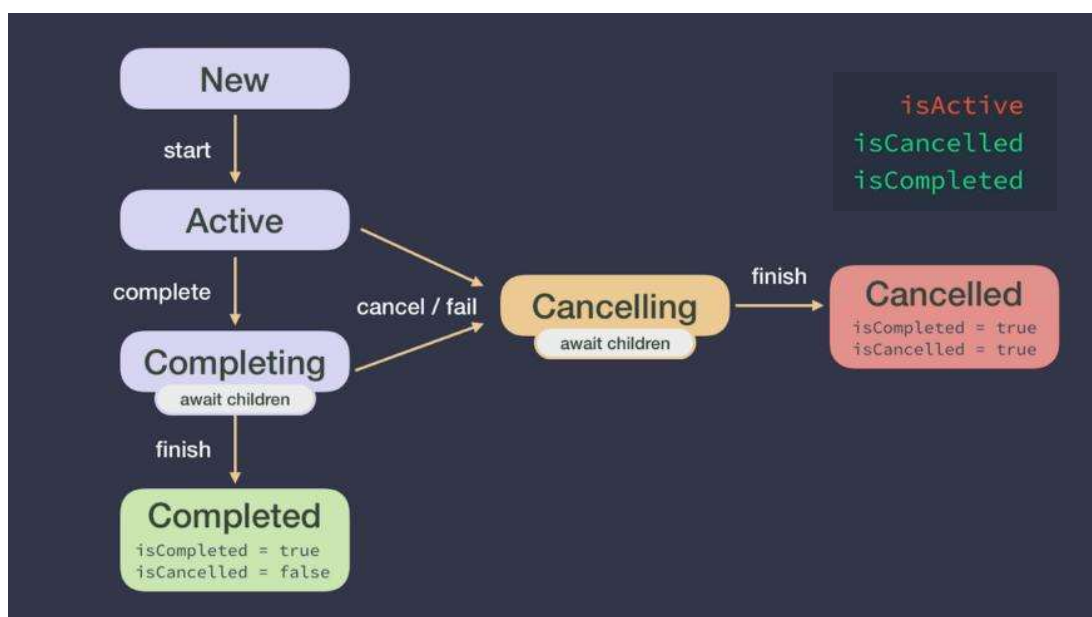
Job

- A **Job** is a *handle* to a coroutine.
 - Coroutine builders (`launch` or `async`) returns a **Job** instance that *uniquely identifies* the coroutine and manages its lifecycle.
- You can also pass a **Job** to a **CoroutineScope** to keep a handle on its lifecycle. Otherwise, `default` Job created

```
val scope = CoroutineScope(Job())
```
- Coroutines form a *hierarchy* using *parent-child* relationships among **Jobs**.

5

Job lifecycle



6

CoroutineContext

The `CoroutineContext` is a set of elements that define the behavior of a coroutine:

- `Job` — controls the lifecycle of the coroutine.
- `CoroutineDispatcher` — dispatches work to the appropriate thread.
- `CoroutineName` — name of the coroutine, useful for debugging.
- `CoroutineExceptionHandler` — handles uncaught exceptions.

CoroutineContext

`CoroutineDispatcher` → Threading
`Job` → Lifecycle
`CoroutineExceptionHandler`
`CoroutineName`

Defaults

`CoroutineDispatcher` → `Dispatchers.Default`
`Job` → No parent Job
`CoroutineExceptionHandler` → None
`CoroutineName` → "coroutine"

7

What's the CoroutineContext of a new coroutine?

- A **new instance** of `Job` will be created, allowing us to control its lifecycle.
- The rest of the elements will be **inherited** from the parent's `CoroutineContext`

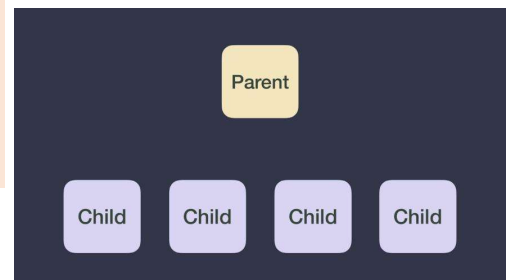
8

Task Hierarchy

- Since a `CoroutineScope` can create coroutines and you can create more coroutines inside a coroutine, an implicit task hierarchy is created.

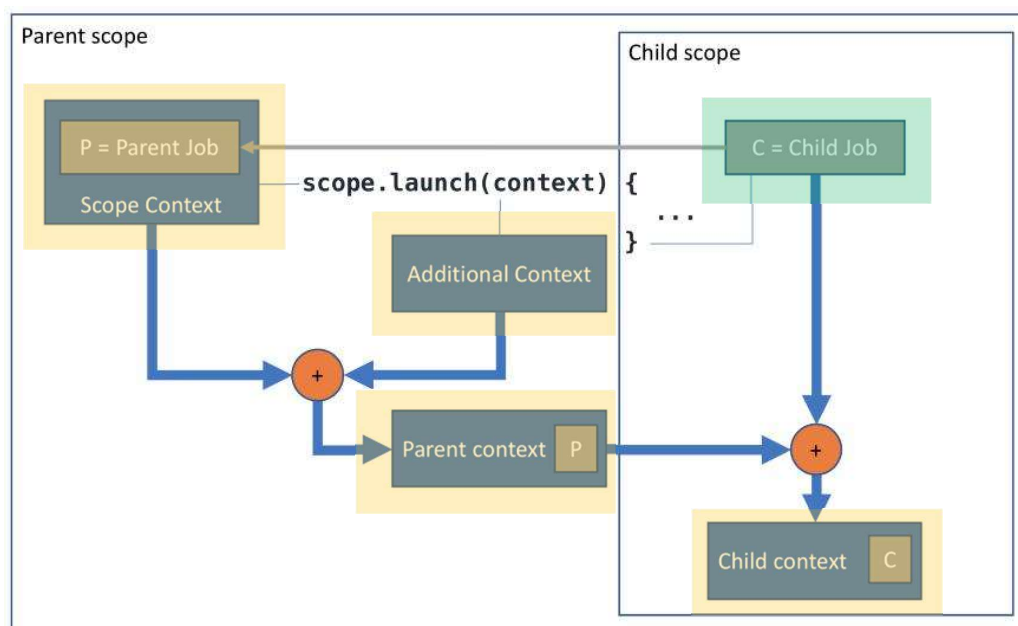
```
val scope = CoroutineScope(Job() + Dispatchers.Main)
val job = scope.launch {
    // New coroutine with CoroutineScope as a parent
    val result = async {
        // New coroutine that has the coroutine
        // started by launch as a parent
    }.await()
}
```

The parent can be either a `CoroutineScope` or another coroutine.



9

Parent Scope vs Child Scope



10

Parent CoroutineContext explained

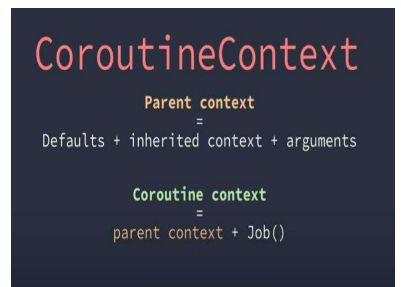
- Child's parent `CoroutineContext` can be different from that of the parent:

$$\text{Parent context} = \text{Defaults} + \text{inherited } \text{CoroutineContext} + \text{arguments}$$



Where:

- Some elements have **default** values: `Dispatchers.Default` is the default of `CoroutineDispatcher` and "coroutine" the default of `CoroutineName`.
- The **inherited** `CoroutineContext` is the `CoroutineContext` of the `CoroutineScope` or coroutine that created it.
- Arguments** passed in the coroutine builder will take precedence over those elements in the inherited context.



11

CoroutineContext of the Parent

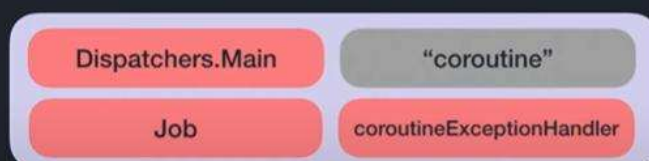


12

CoroutineContext of the Parent

```
// Defaults: Dispatchers.Default, "coroutine"
val scope = CoroutineScope(
    Job() + Dispatchers.Main + coroutineExceptionHandler
)
```

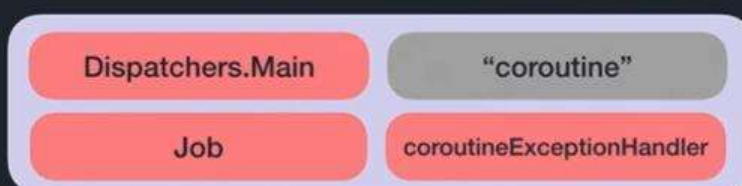
Parent context



Every coroutine started by this `CoroutineScope` will have at least those elements in the `CoroutineContext`. `CoroutineName` is gray because it comes from the default values.

13

Parent context

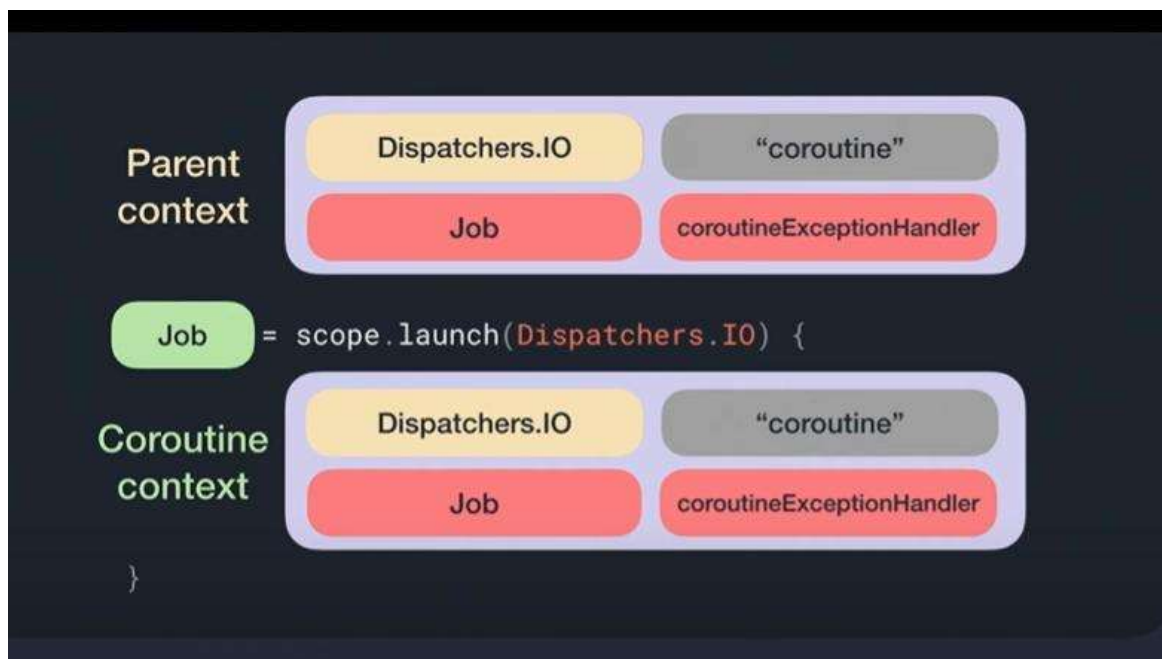


```
val job = scope.launch(Dispatchers.IO) {
    // CoroutineContext?
```

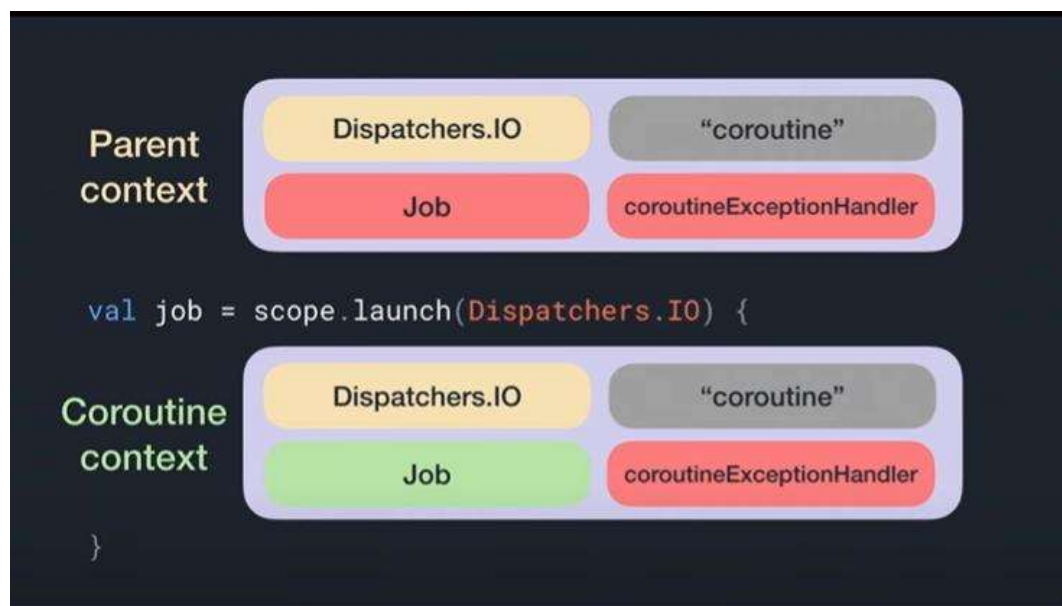
New coroutine context = parent `CoroutineContext` + `Job()`

```
}
```

14



15



The Job in the CoroutineContext and in the parent context will never be the same instance as a new coroutine always get a new instance of a Job

16

Exception Propagation

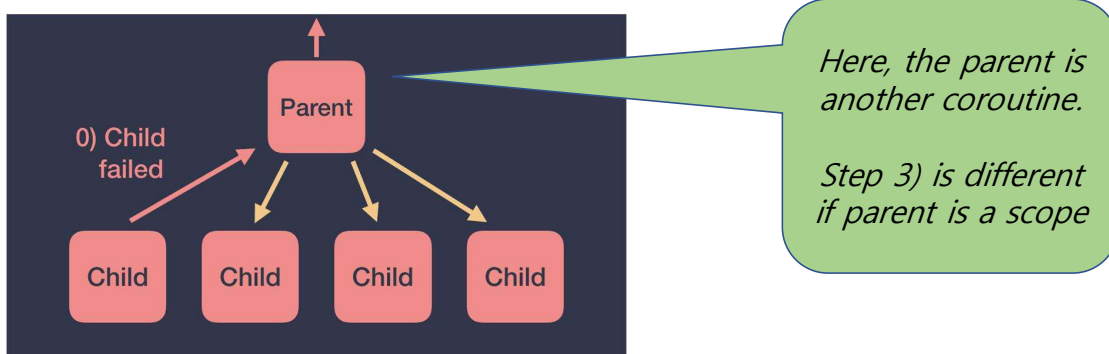
- An **uncaught exception**, instead of being re-thrown, is “**propagated up the job hierarchy**”.



17

Exception Propagation

- This exception propagation leads to the failure of the parent Job and the cancellation of all the Jobs of its children.
- The exception will reach the root of the hierarchy and all the coroutines that the **CoroutineScope** started will get cancelled too.



18

Exception Re-throwing vs. Propagation

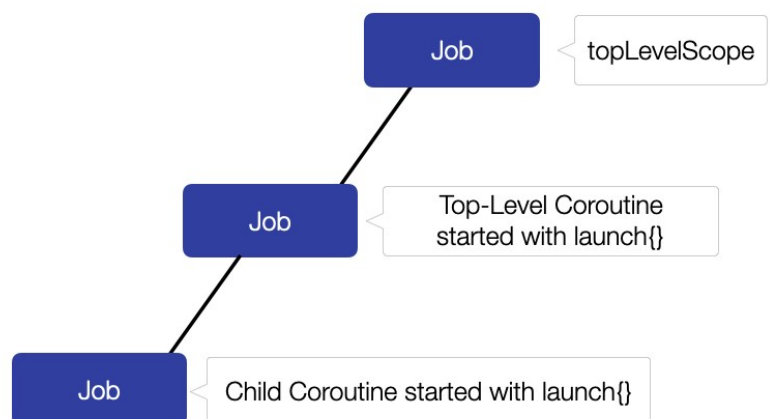
- In Kotlin, functions by default **re-throw** all the exceptions that were not caught inside them.
- Therefore, the exception from the `failingMethod` can be caught in the parent `try-catch` block.

```
fun someMethod() {  
    try {  
        val failingData = failingMethod()  
    } catch (e: Exception) {  
        // handle exception  
    }  
}  
  
fun failingMethod() { throw RuntimeException() }
```

19

Coroutines Parent-Child Relationship

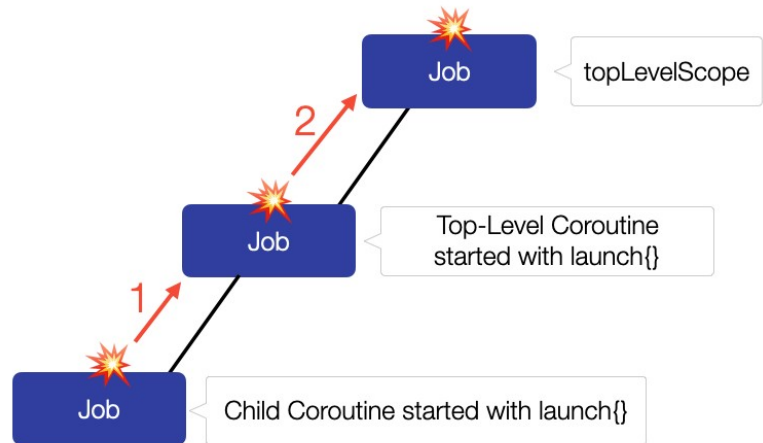
```
fun main() {  
    val scope = CoroutineScope(Job())  
    scope.launch {  
        try {  
            launch {  
                throw RuntimeException("...")  
            }  
        } catch (ex: Exception) {  
            // do something ...  
        }  
    }  
    Thread.sleep(100)  
}
```



20

Exception Propagation up to ...

```
fun main() {  
    val scope = CoroutineScope(Job())  
    scope.launch {  
        try {  
            launch {  
                throw RuntimeException("...")  
            }  
        } catch (ex: Exception) {  
            // do something ...  
        }  
    }  
    Thread.sleep(100)  
}
```



21

Exception Re-Thrown vs. Propagation

```
fun main() {  
    try {  
        failingMethod()  
    } catch (ex: Exception) {  
        println("Caught: $ex")  
    }  
}
```

Caught: java.lang.RuntimeException: oops

```
fun main() = runBlocking<Unit> {  
    try {  
        launch {  
            failingMethod()  
        }  
    } catch (ex: Exception) {  
        println("Caught: $ex")  
    }  
}
```

Useless!

Exception in thread "main" java.lang.RuntimeException: oops
at com.org.androidtestingkt.coroutines. ...

22

Exception Propagation: Root is not Scope

```
@Test(expected = RuntimeException::class)
fun `Uncaught exceptions propagate`() = runBlocking {
    val job = launch {
        println("1. Exception thrown inside launch")
        throw RuntimeException()
    }
    println("2. Wait for child to finish")
    job.join()
    println("3. Joined failed job: Unreachable code")
}
```

2. Wait for child to finish
1. Exception thrown inside launch

23

Exception Propagation: Root is Scope

```
@Test
fun `Uncaught exceptions propagate`() = runBlocking {
    val scope = CoroutineScope(Job())
    val job = scope.launch {
        println("1. Exception thrown inside launch")
        // handled by Thread.defaultUncaughtExceptionHandler
        throw RuntimeException()
    }
    println("2. Wait for child to finish")
    job.join()
    println("3. Joined failed job: Now reachable code")
}
```

2. Wait for child to finish
1. Exception thrown inside launch
Exception in thread "DefaultDispatcher-worker-1 ...
...
3. Joined failed job: Now reachable code 24

Failures in a Scope

- The failure of a child cancels the parent and other children.

```
val parentJob = launch {  
    launch {  
        throw RuntimeException("oops")  
    }.invokeOnCompletion { ex ->  
        println("child1: $ex")  
    }  
    launch {  
        delay(100)  
    }.invokeOnCompletion { ex ->  
        println("child2: $ex")  
    }  
}  
parentJob.invokeOnCompletion {  
    println("isCancelled = ${parentJob.isCancelled}")  
}  
parentJob.join()
```

child1: java.lang.RuntimeException: oops
child2: kotlinx.coroutines.JobCancellationException: ...
isCancelled = true

25

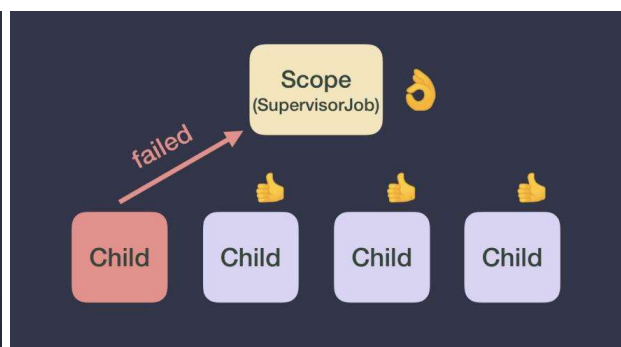
SupervisorJob to the rescue

- A *SupervisorJob* won't cancel itself or the rest of its children

SupervisorJob

```
val scope = CoroutineScope(SupervisorJob())
```

The **failure** or **cancellation** of a child
doesn't affect other children



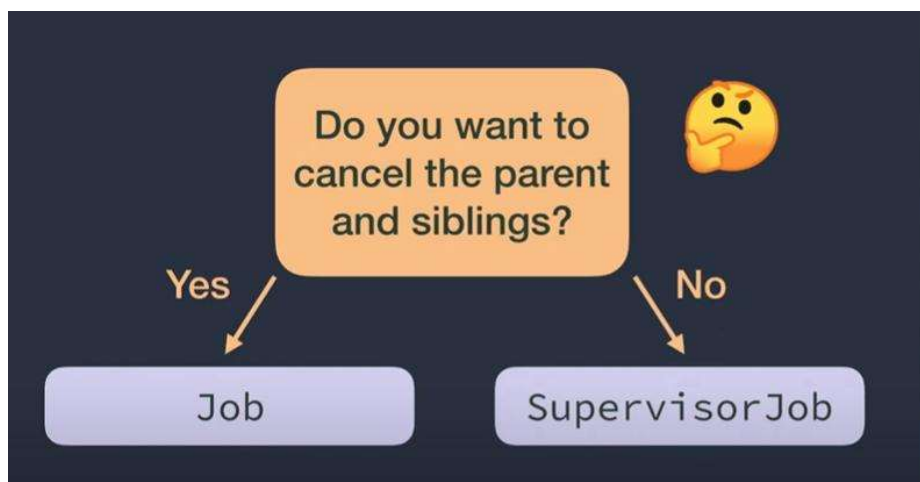
26

Review of SupervisorJob

- A `CoroutineScope` can have a `SupervisorJob` that changes how the `CoroutineScope` deals with exceptions.
- However, when the **parent of a coroutine is another coroutine**, the parent `Job` will **always** be of type `Job`.

27

What to choose?



28

WATCH OUT #2

SupervisorJob **only** works if it is
the coroutine's **direct** parent

29

Watch out quiz! Who's my parent?

- Given the following snippet of code, can you identify what kind of **Job** “child 1” has as a parent?

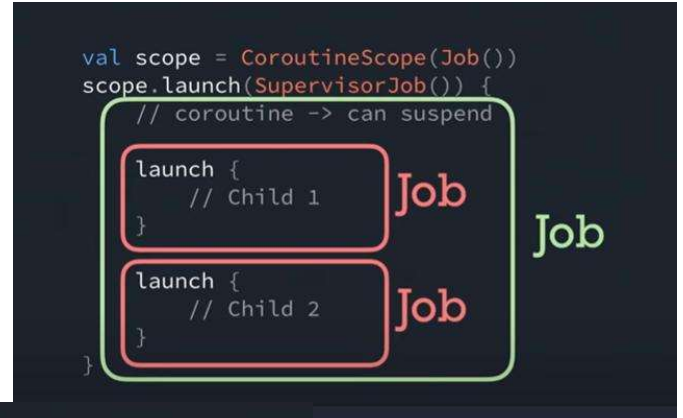
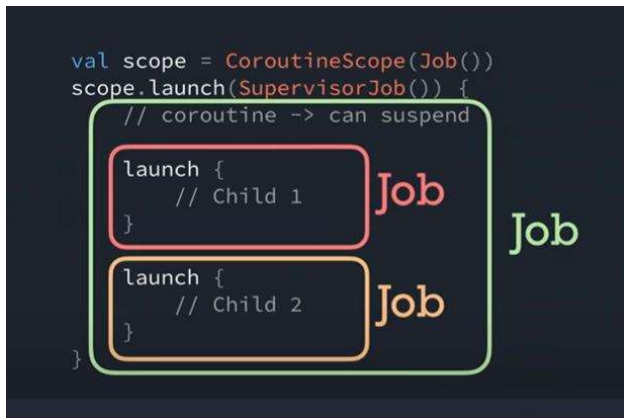
```
val scope = CoroutineScope(Job())
scope.launch(SupervisorJob()) {
    // new coroutine -> can suspend
    launch {
        // Child 1
    }
    launch {
        // Child 2
    }
}
```

```
val scope = CoroutineScope(Job())
scope.launch(SupervisorJob()) {
    // coroutine -> can suspend
    launch {
        // Child 1
    }
    launch {
        // Child 2
    }
}
```

Job

Job

30



31

Remember that a SupervisorJob only works when it's part of a scope:

- Either created as a direct child of either `supervisorScope` or `CoroutineScope(SupervisorJob())`.
- Passing a `SupervisorJob` as a parameter of a coroutine builder will **not** have the desired effect.
- If any child throws an exception, that `SupervisorJob` won't either propagate up in the hierarchy or rethrow the exception. Instead, it delegates the exception to `CoroutineExceptionHandler`, if exists, or `Thread.uncaughtExceptionHandler`.

32


```

val scope = CoroutineScope(Job())
scope.launch {
    supervisorScope {
        launch {
            // Child 1
        }
        launch {
            // Child 2
        }
    }
}

```

Job

Job

```

val scope = CoroutineScope(Job())
scope.launch {
    supervisorScope {
        launch {
            // Child 1
        }
        launch {
            // Child 2
        }
    }
}

```

SupervisorJob

Job

33

Parent

```

val scope = CoroutineScope(Job())
val sharedJob = SupervisorJob()

scope.launch(sharedJob) {
    // Child 1
}

scope.launch(sharedJob) {
    // Child 2
}

```

Job

Job

```

val viewPresenterScope = CoroutineScope(
    SupervisorJob()
)
fun refreshData() {
    viewPresenterScope.launch {
        // Load payments
    }
    viewPresenterScope.launch {
        // Load transactions
    }
}

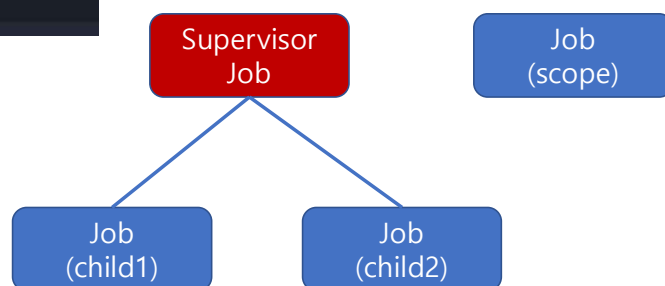
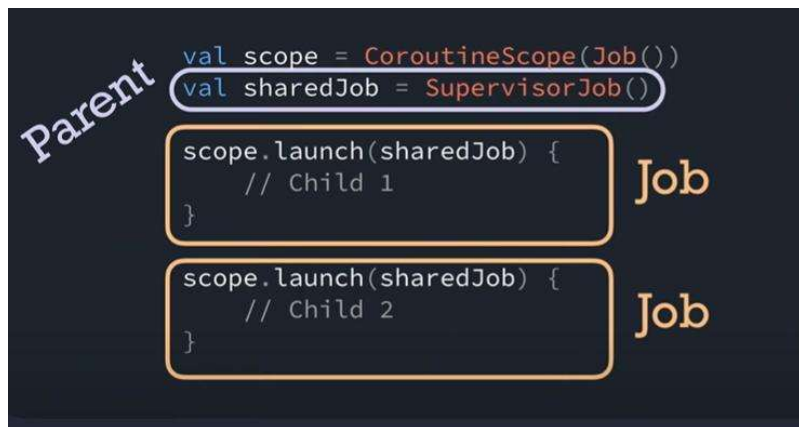
```

Parent

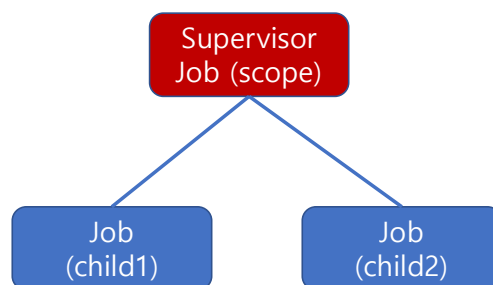
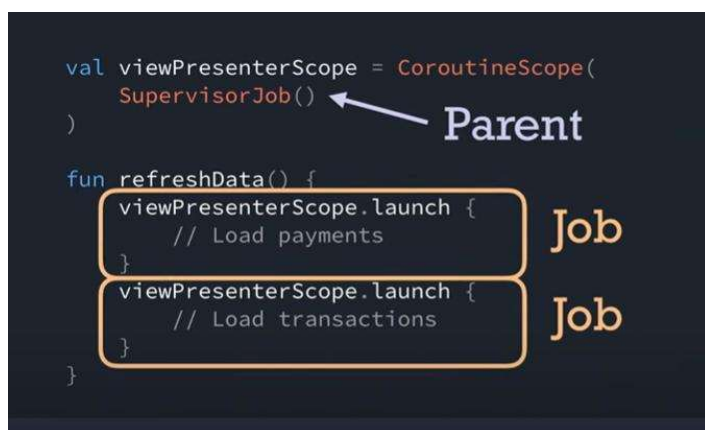
Job

Job

34



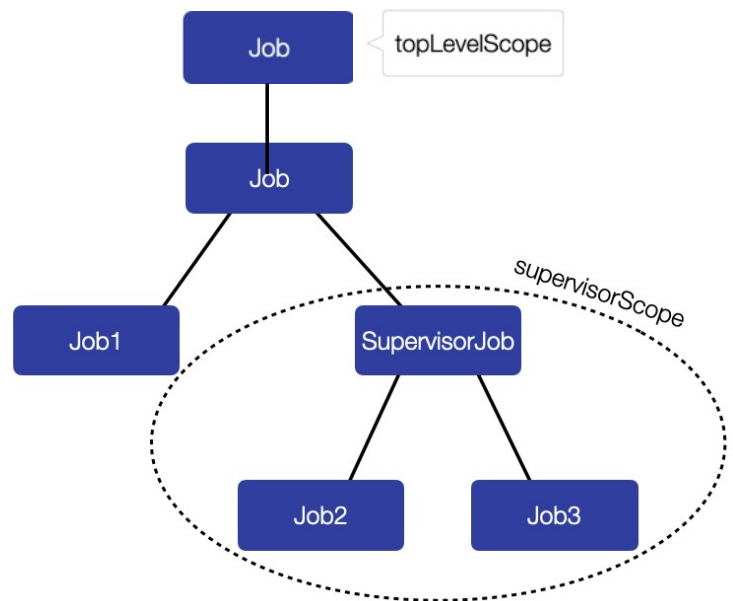
35



36

Exception Handling properties of supervisorScope{}

```
val scope = CoroutineScope(Job())
scope.launch {
    val job1 = launch {
        println("starting Coroutine 1")
    }
    supervisorScope {
        val job2 = launch(ehandler) {
            println("starting Coroutine 2")
        }
        val job3 = launch {
            println("starting Coroutine 3")
        }
    }
}
```



37

WATCH OUT #n

Check the implementation of
predefined scopes!

e.g. viewModelScope or lifecycleScope

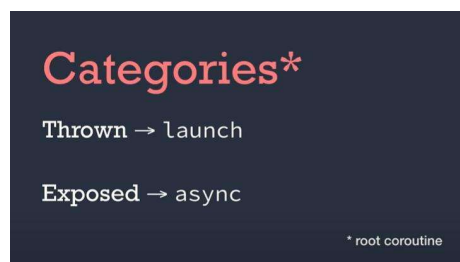
38

Dealing with Exceptions

- `try/catch`
- `runCatching` (which uses `try/catch` internally)
- `CoroutineExceptionHandler`

Recall that uncaught exceptions will always be thrown.

However, different coroutines builders treat exceptions in different ways.



39

Coroutine Builder `launch` Behavior

- With `launch`, ***exceptions will be thrown as soon as they happen.*** Therefore, you can wrap the code that can throw exceptions inside a `try/catch`, like in this example:


```
launch {  
    try {  
        println("1. Exception thrown inside launch")  
        throw RuntimeException()  
    } catch (ex: Exception) {  
        println("Exception ${ex.javaClass.simpleName} caught ...")  
    }  
}
```

inside launch

40

Coroutine Builder `async` Behavior

Root coroutines:

Coroutines that are a direct child of a  `CoroutineScope` or `supervisorScope`



41

Coroutine Builder `async` Behavior

- **Root coroutines:** coroutines that are a direct child of a `CoroutineScope` or `supervisorScope`
- When `async` is used as a root coroutine, **exceptions are not thrown automatically, instead, they're thrown when you call `.await()`.**
- To handle exceptions thrown in `async` whenever it's a root coroutine, you can wrap the `.await()` call inside a `try/catch`:

```
supervisorScope {  
    val deferred = async {  
        throw codeThatMayThrowException()  
    }  
    try {  
        deferred.await()  
    } catch (e: Exception) {  
        println("Caught ${e.javaClass.simpleName}")  
    }  
}
```

42

Watch out!

- Notice that we're using a `supervisorScope` to call `async` and `await`.
- But, a `coroutineScope` automatically propagate the exception up in the hierarchy so the catch block won't be called:

```
coroutineScope {  
    try {  
        val deferred = async {  
            codeThatCanThrowExceptions()  
        }  
        deferred.await()  
    } catch (e: Exception) {  
        // Exception thrown in async WILL NOT be caught here  
        // but propagated up to the scope  
    }  
}
```

43

Furthermore, exceptions that happen in coroutines created by other coroutines will always be propagated regardless of the coroutine builder.

```
val scope = CoroutineScope(Job())  
  
scope.launch {  
    val deferred = async {  
        // If async throws, launch throws  
        throw RuntimeException()  
    }  
}
```



⚠ Exceptions thrown in a `coroutineScope` builder or in coroutines created by other coroutines won't be caught in a `try/catch`!

CoroutineExceptionHandler

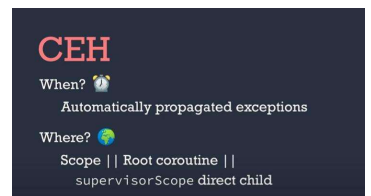
- The `CoroutineExceptionHandler` is an optional element of a `CoroutineContext` allowing you to handle uncaught exceptions.

```
val handler = CoroutineExceptionHandler {  
    context, exception -> println("Caught $exception")  
}
```

45

Exceptions will be caught if these requirements are met:

- **When** 🕒: Automatically propagated exceptions
- **Where** 🌐: If it's in the `CoroutineContext` of a `CoroutineScope` or a root coroutine (direct child of `CoroutineScope` or a `supervisorScope`).



```
val  
scope = CoroutineScope(Job() + handler)  
  
scope.launch {  
    launch {  
        throw Exception("failed")  
    }  
}
```

```
val scope = CoroutineScope(Job())  
  
scope.launch(handler) {  
    launch {  
        throw Exception("failed")  
    }  
}
```

46

```
scope.launch {
    try {
        codeThatCanThrowExceptions()
    } catch(e: Exception) {
        // Handle exception
    }
}
```

```
supervisorScope {
    val deferred = async {
        codeThatCanThrowExceptions()
    }
    try {
        deferred.await()
    } catch(e: Exception) {
        // Handle exception thrown in async
    }
}
```

47

```
coroutineScope {
    val deferred = async {
        codeThatCanThrowExceptions()
    }
    try {
        deferred.await()
    } catch(e: Exception) {
        // This WON'T be called! 😞
    }
}
```



```
scope.launch {
    val result = runCatching {
        codeThatCanThrowExceptions()
    }

    if (result.isSuccess) {
        // Happy path
    } else {
        // Sad path
    }
}
```

48


```
val handler = CoroutineExceptionHandler {
    _, exception -> println("Caught $exception")
}


val scope = CoroutineScope(Job() + handler)

scope.launch {
    launch {
        throw Exception("Failed coroutine")
    }
}
```

```
val handler = CoroutineExceptionHandler {
    _, exception -> println("Caught $exception")
}

val scope = CoroutineScope(Job())

scope.launch {
    launch(handler) {
        throw Exception("Failed coroutine")
    }
}
```




```
val handler = CoroutineExceptionHandler {
    _, exception -> println("Caught $exception")
}

scope.launch {
    supervisorScope {
        launch(handler) {
            throw Exception("Failed coroutine")
        }
    }
}
```

```
val handler = CoroutineExceptionHandler {
    _, exception -> println("Caught $exception")
}

scope.launch {
    coroutineScope {
        launch(handler) {
            throw Exception("Failed coroutine")
        }
    }
}
```



Coroutine builders

coroutineScope

- It **inherits the caller's CoroutineContext** and supports Structured Concurrency.
- It doesn't propagate exceptions from its children but **re-throws** them instead.
- It **cancels all other children if one of them fails**.

supervisorScope

- It **inherits the caller's CoroutineContext** and supports Structured Concurrency.
- It doesn't propagate exceptions from its children. Call CEH if exists. Otherwise call default uncaught exception handler.
- If one of the coroutines inside fails, the others are **not cancelled**.
- Coroutines created inside become **top-level coroutines**.

51

Summary

- Dealing with exceptions gracefully in your application is important to have a good user experience, even when things don't go as expected.
- Remember to use **SupervisorJob** when you want to avoid propagating cancellation when an exception happens, and **Job** otherwise.
- Uncaught exceptions will be propagated, catch them to provide a great UX!

52