



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

(Review) 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"
```

4

(Review)

What's the CoroutineContext of a new coroutine?

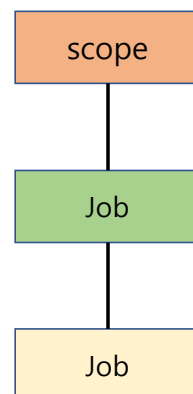
- A **new instance** of `Job` will be created, allowing us to control its lifecycle.
- The rest of the elements will be **inherited** from the parent's `CoroutineContext`

5

(Review) Task Hierarchy

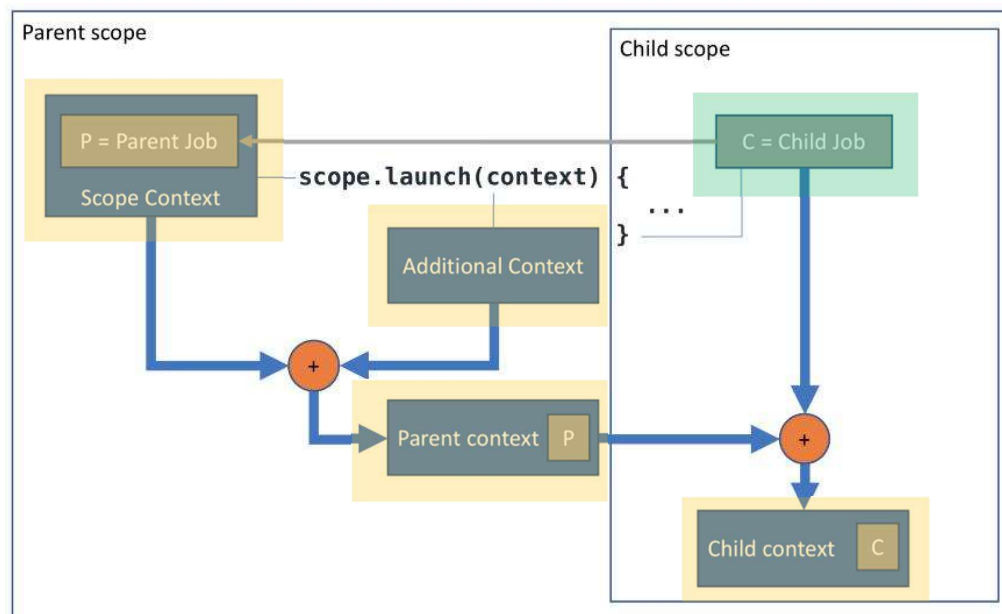
- Since a `CoroutineScope` can create coroutines and you can create more coroutines inside a coroutine, an implicit task hierarchy is created.

```
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()
}
```



6

(Review) Parent Scope vs Child Scope



7

Exception Propagation

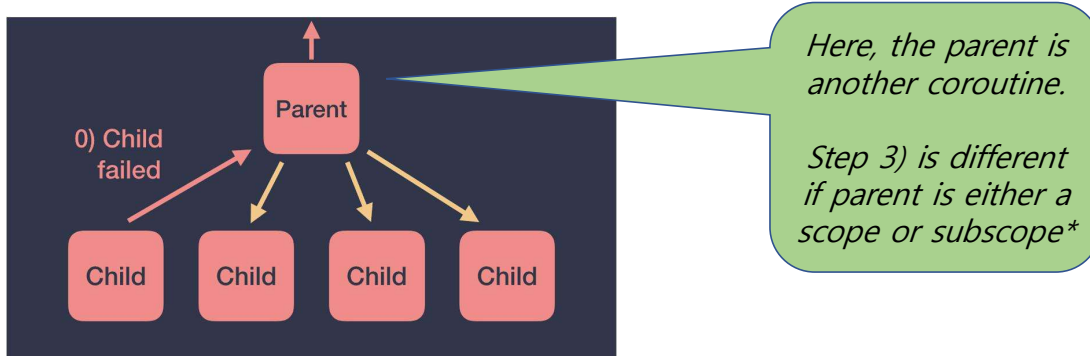
- An **uncaught exception**, instead of being re-thrown, is “**propagated up the job hierarchy**”.



8

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 (**default behavior**).



9

Exception Re-throwing vs. Propagation

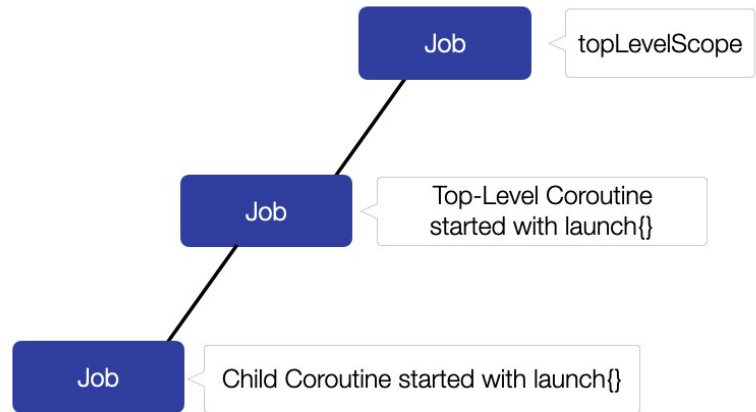
- In Kotlin, functions by default **re-throw** all the exceptions that were not caught inside them.
- Therefore, the exception from the `failingMethod` can be caught in the parent `try-catch` block.

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

10

Coroutines Parent-Child Relationship

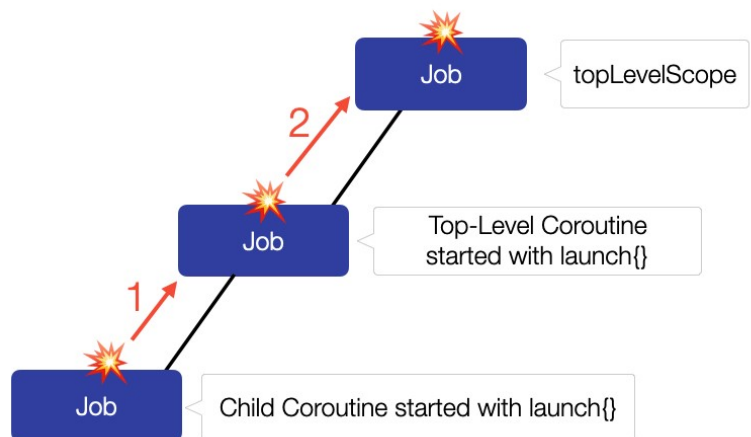
```
fun main() {  
    val scope = CoroutineScope(Job())  
    scope.launch {  
        try {  
            launch {  
                throw RuntimeException("...")  
            }  
        } catch (ex: Exception) {  
            // do something ...  
        }  
    }  
    Thread.sleep(100)  
}
```



11

Exception Propagation up to ...

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



12

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. ...

13

Exception Propagation: Root is not a 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

14

Exception Propagation: Root is a 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 15

Failures in a Scope

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

- The failure of a child cancels the parent and child's other siblings.

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


SupervisorJob to the rescue

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



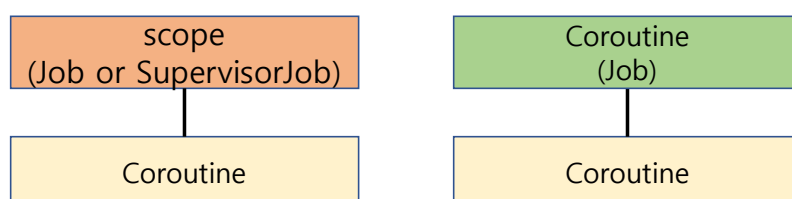
17

Review of SupervisorJob

- A `CoroutineScope` can have a `SupervisorJob` that changes how the `CoroutineScope` deals with exceptions.

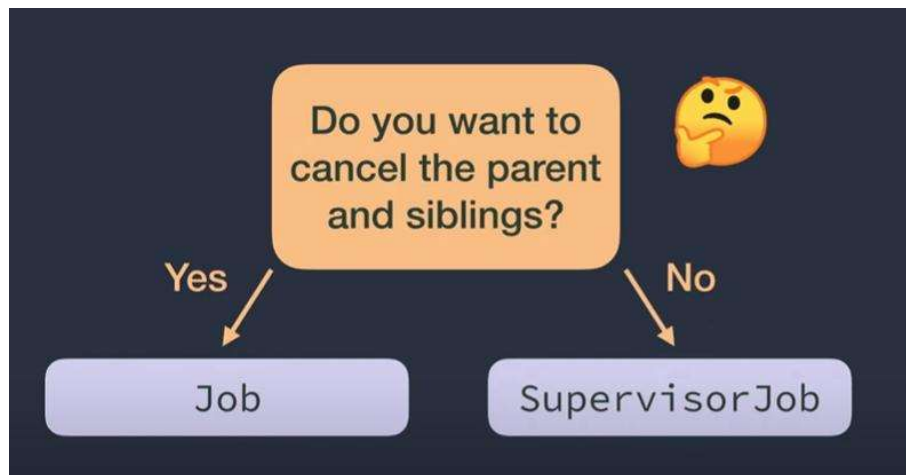
```
val scope = CoroutineScope(SupervisorJob())
```

- However, when the **parent of a coroutine is another coroutine**, the parent `Job` will **always** be of type `Job`.



18

What to choose?



19

Somewhat misleading statement ... I think

WATCH OUT #2

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

20

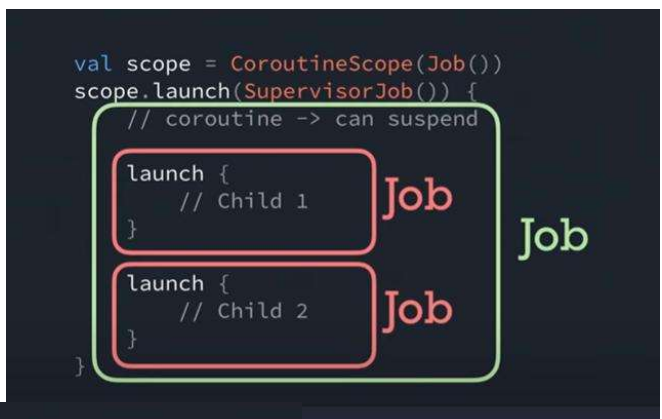
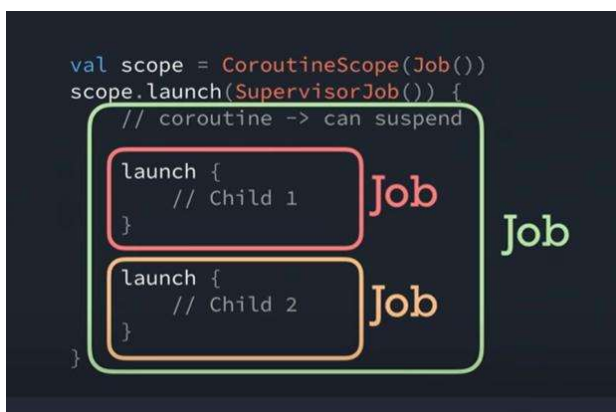
Watch out quiz! Who's my parent?

- 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
    }
}
```



21



22

SupervisorJob protects a failed child's siblings only if it is a failed coroutine's direct parent

- Coroutines that are created as a direct child of either `supervisorScope` or `CoroutineScope(SupervisorJob())`.
- Passing a `SupervisorJob` as a parameter of a coroutine builder *may* **not** have your desired effect.



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

23

Exception handling Behavior of SupervisorJob and top-level Scopes

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`.

The same is true with any top-level scopes and `supervisorScope`.

24

```

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

```

Coroutines are created as a direct child of `supervisorScope`

```

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

```

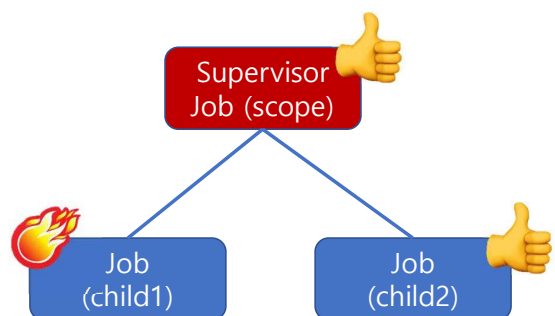
25

```

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

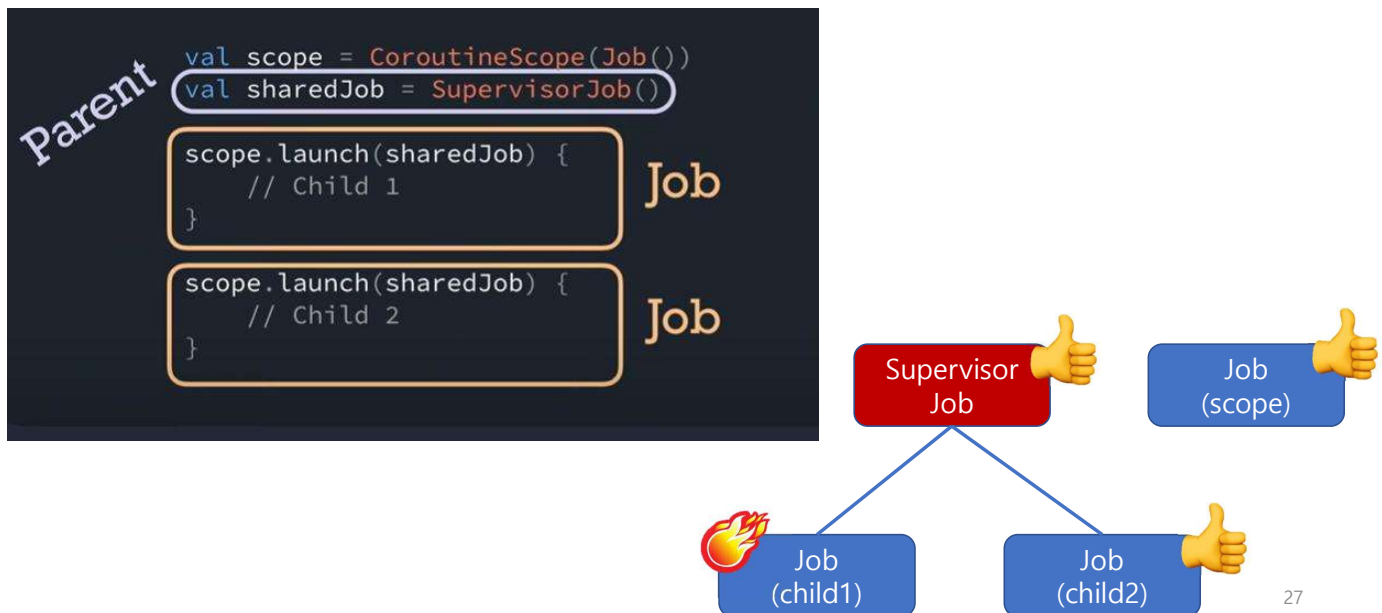
```

Coroutines are created as a direct child of `CoroutineScope(SupervisorJob())`



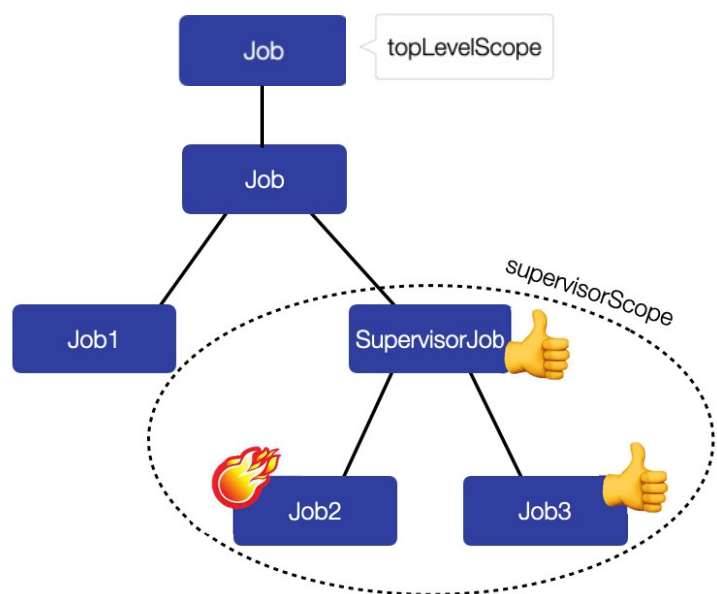
26

Coroutines are created and adopted by `SupervisorJob()`



Exception Handling properties of supervisorScope

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



WATCH OUT #n

Check the implementation of
predefined scopes!

e.g. `viewModelScope` or `lifecycleScope`

29

Dealing with Exceptions

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

Exceptions

1. `try/catch`
2. `runCatching`
3. `CoroutineExceptionHandler`

Recall that uncaught exceptions will always be propagated by default.

However, different coroutines builders treat exceptions in different ways.

Uncaught Exceptions →

Categories*

Thrown → `launch` (or `Unhandled`)

Exposed → `async`

* root coroutine

30

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

31

Coroutine Builder: `async` Behavior

Root coroutines:

Coroutines that are a direct child of a



`CoroutineScope` or `supervisorScope`



32

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}")  
    }  
}
```

33

Watch out!

- Notice that we're using a `supervisorScope` to call `async` and `await`.
- But, a `coroutineScope` automatically rethrows the exception so the catch block ~~won't be called~~: => *will be called, but still exception propagates*

```
coroutineScope {  
    try {  
        val deferred = async {  
            codeThatCanThrowExceptions()  
        }  
        deferred.await()  
    } catch (e: Exception) {  
        // Exception thrown in async WILL NOT be caught here  
        // but still will be rethrown by the coroutineScope  
    }  
}
```

34

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! **`async`: caught but still propagates!**

33

CoroutineExceptionHandler

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

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

Exceptions will be caught by the CEH if these requirements are met:

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

CEH

When? 🕒

Automatically propagated exceptions

Where? 🌐

Scope || Root coroutine ||
supervisorScope direct child

```
val
scope=CoroutineScope(Job() + handler)

scope.launch {
    launch {
        throw Exception("failed")
    }
}
```

```
val scope = CoroutineScope(Job())

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

37

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

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

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

```

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

```

```

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

```



```

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")
        }
    }
}

```



41

Coroutine builders

coroutineScope

- It inherits the caller's [CoroutineContext](#) and supports Structured Concurrency.
- It doesn't propagate exceptions from its children. Call CEH if exists. Otherwise, **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 child coroutines inside fails, the others are **not cancelled**.
- Coroutines created inside become **root coroutines**.

42

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!