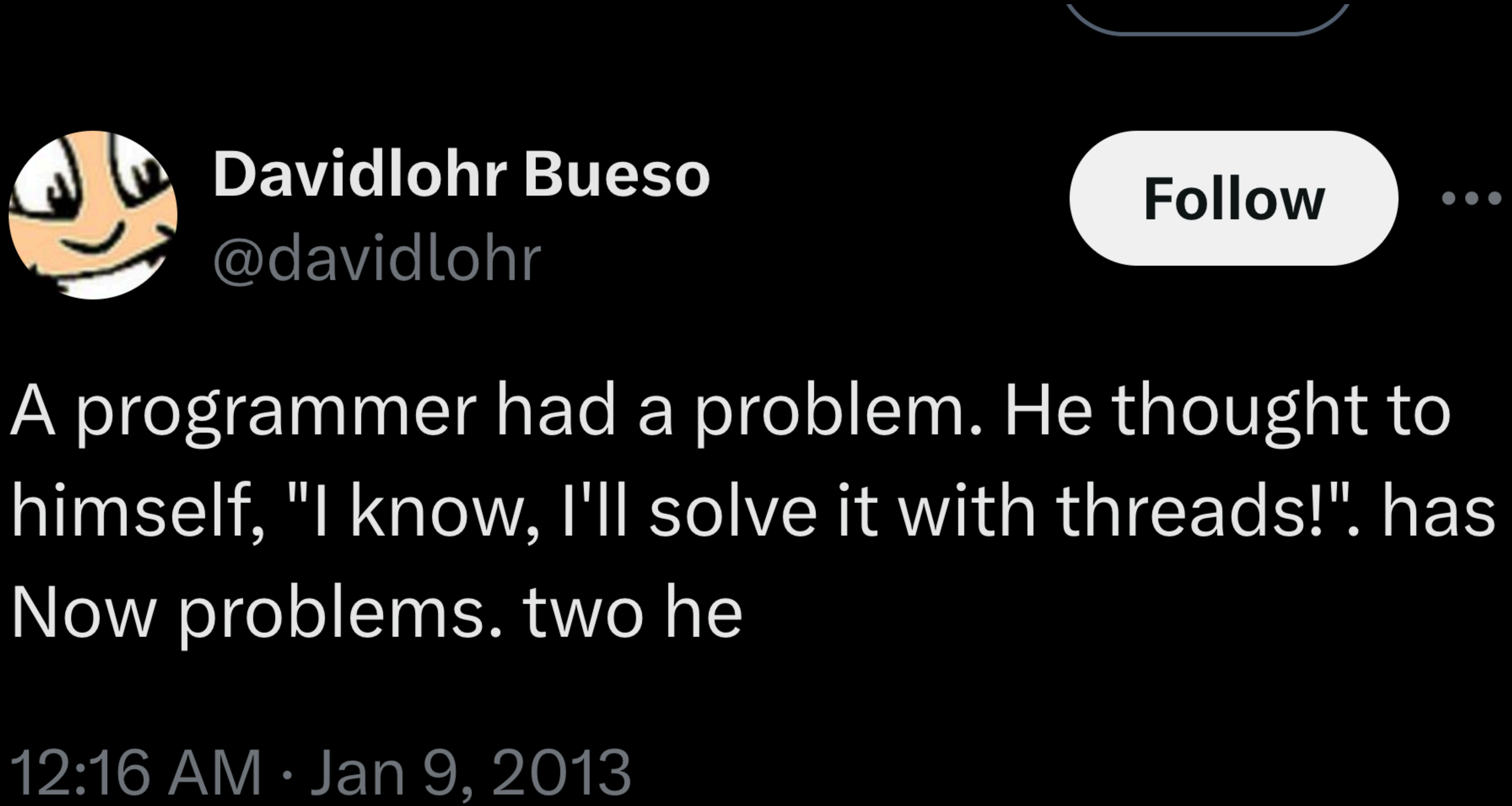
STRUCTURED CONCURRENCY

JESPER HOLMBERG

CADEC 2025.01.23 & 2025.01.29 | CALLISTAENTERPRISE.SE

CALLISTA

CONCURRENCY IS HARD



Davidlohr Bueso
@davidlohr

A programmer had a problem. He thought to himself, "I know, I'll solve it with threads!". has Now problems. two he

12:16 AM · Jan 9, 2013

CONCURRENCY CONSTRUCTS GALORE

- We have seen many concurrency constructs: callbacks, threads, futures, executors, …
- It's easier than ever to create massive amounts of *threads*: Go (goroutines), Kotlin (coroutines), Java (virtual threads), …
- We need good constructs to keep this potential chaos under control
- Structured concurrency is a new(ish) alternative: today we'll look at how it can be used in Java and Kotlin

STRUCTURED CONCURRENCY

Wikipedia:

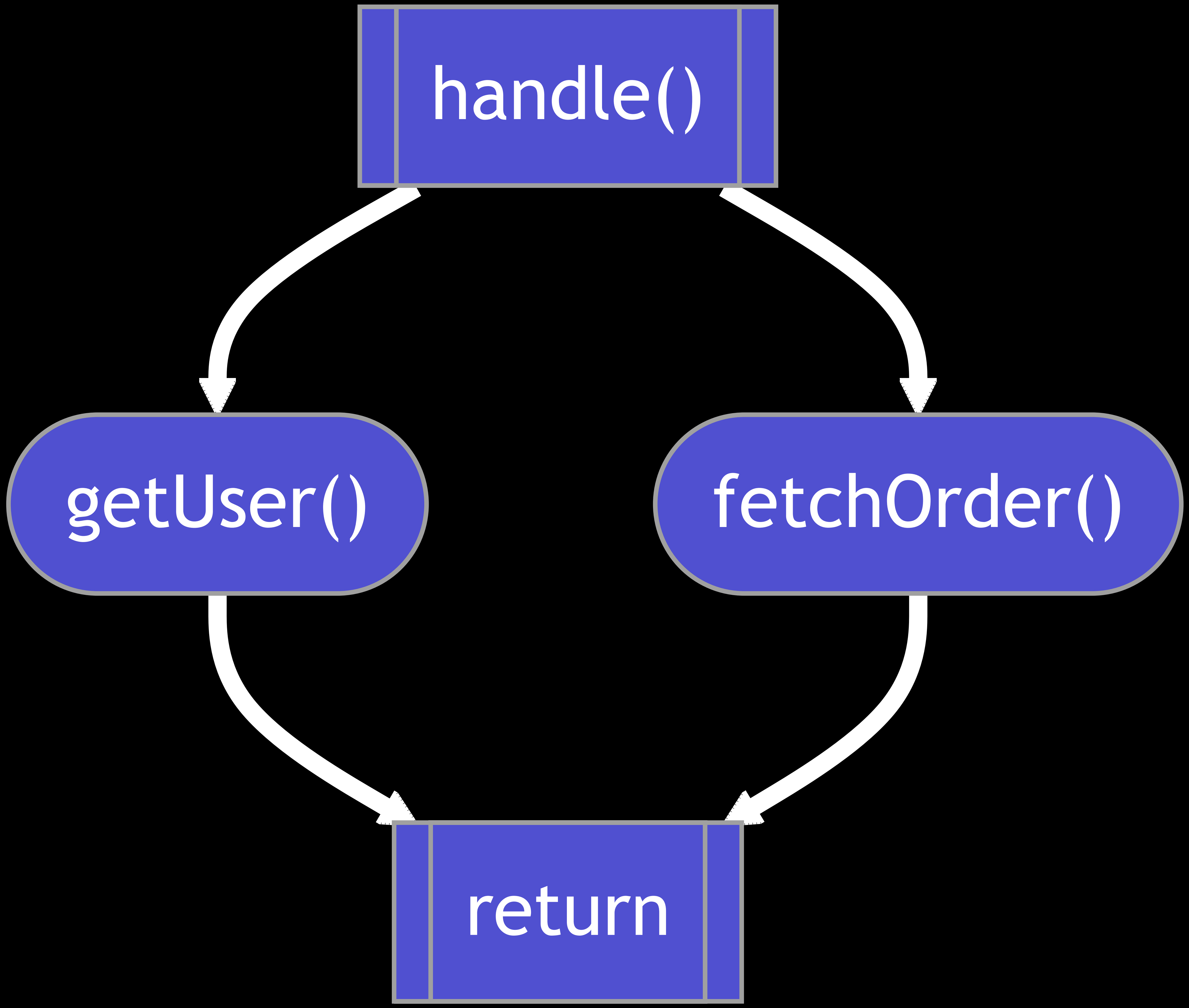
The encapsulation of concurrent threads with control flow constructs that have clear entry and exit points and ensure that all spawned threads have completed before exit.

STRUCTURED CONCURRENCY HISTORY

- Martin Sustrik in C library *libdill (2016)*
- Popularised by Nathaniel J. Smith (Python) in *Notes on structured concurrency, or: Go statement considered harmful (2018)*
- Term picked up by Kotlin designers for their coroutine implementation *(2018)*
- Library implementations now exist in many languages
- Java 23 includes the third preview of JEP 480: Structured Concurrency

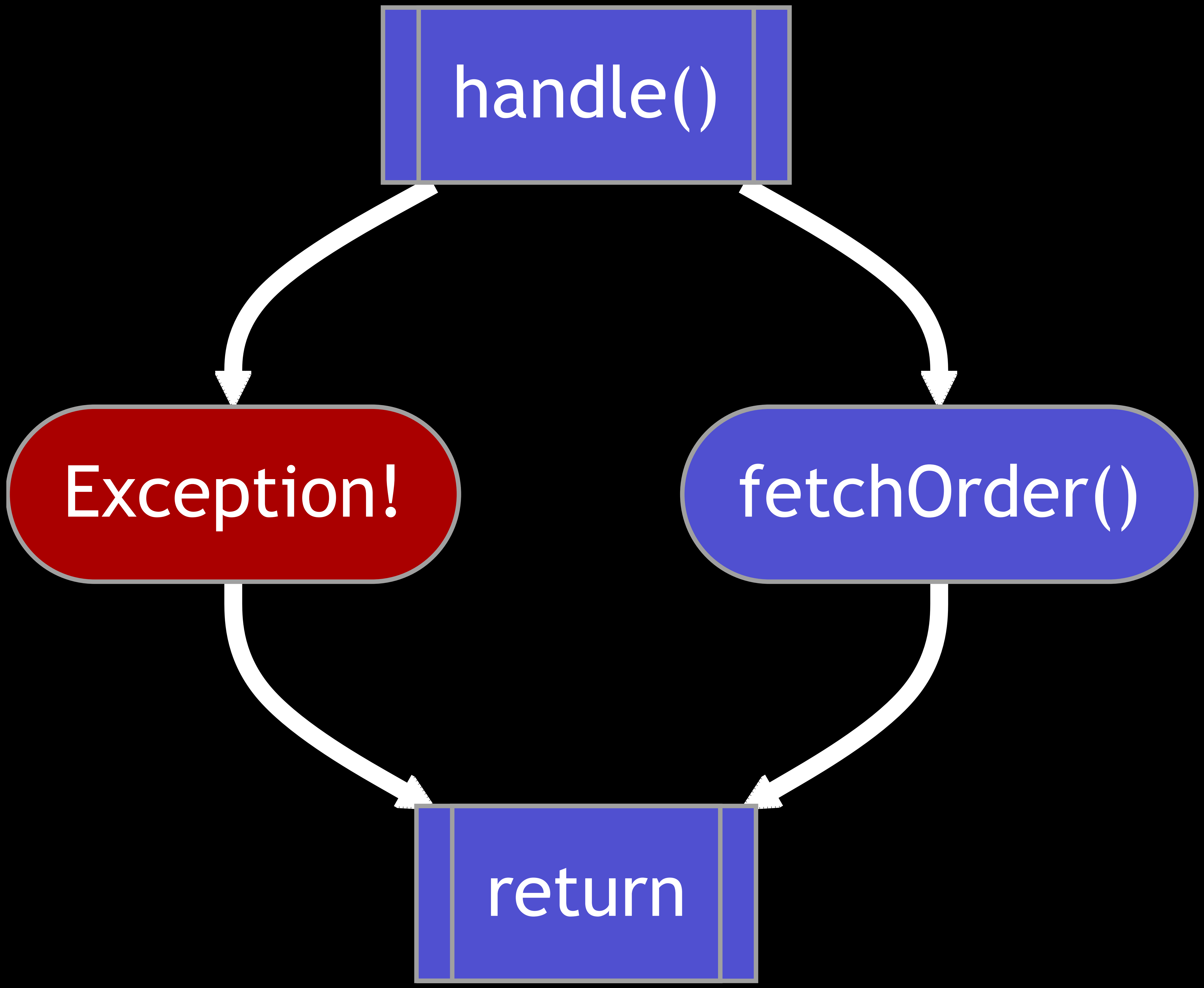
JAVA CONCURRENCY EXAMPLE

```
Response handle() {
  var user = executorService.submit(() -> findUser());
  var order = executorService.submit(() -> fetchOrder());
  var theUser = user.get(); // Join first thread
  var theOrder = order.get(); // Join second thread
  return new Response(theUser, theOrder);
}
```



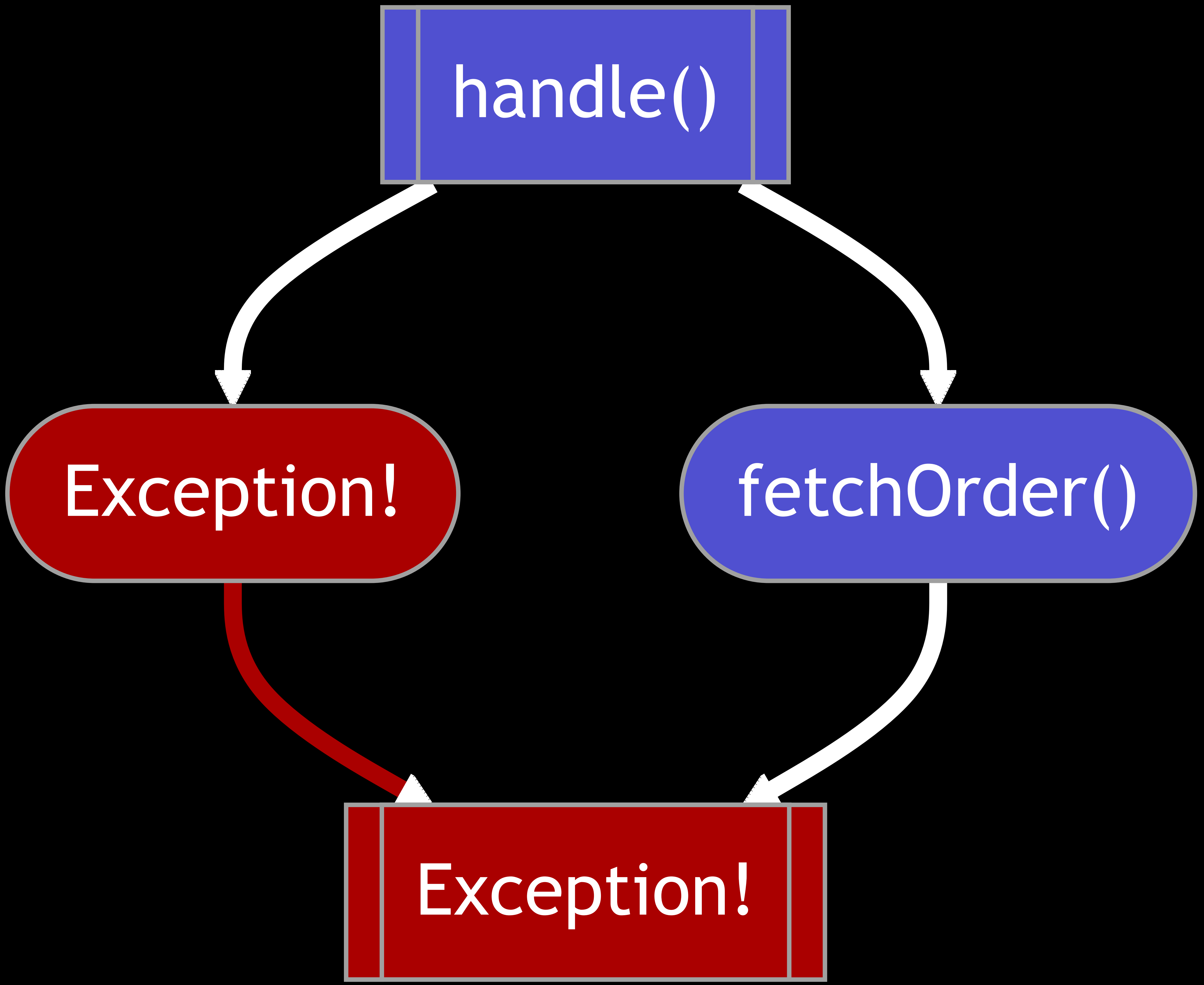
JAVA CONCURRENCY EXAMPLE

```
Response handle() {
  var user = executorService.submit(() -> findUser());
  var order = executorService.submit(() -> fetchOrder());
  var theUser = user.get(); // Join first thread
  var theOrder = order.get(); // Join second thread
  return new Response(theUser, theOrder);
}
```



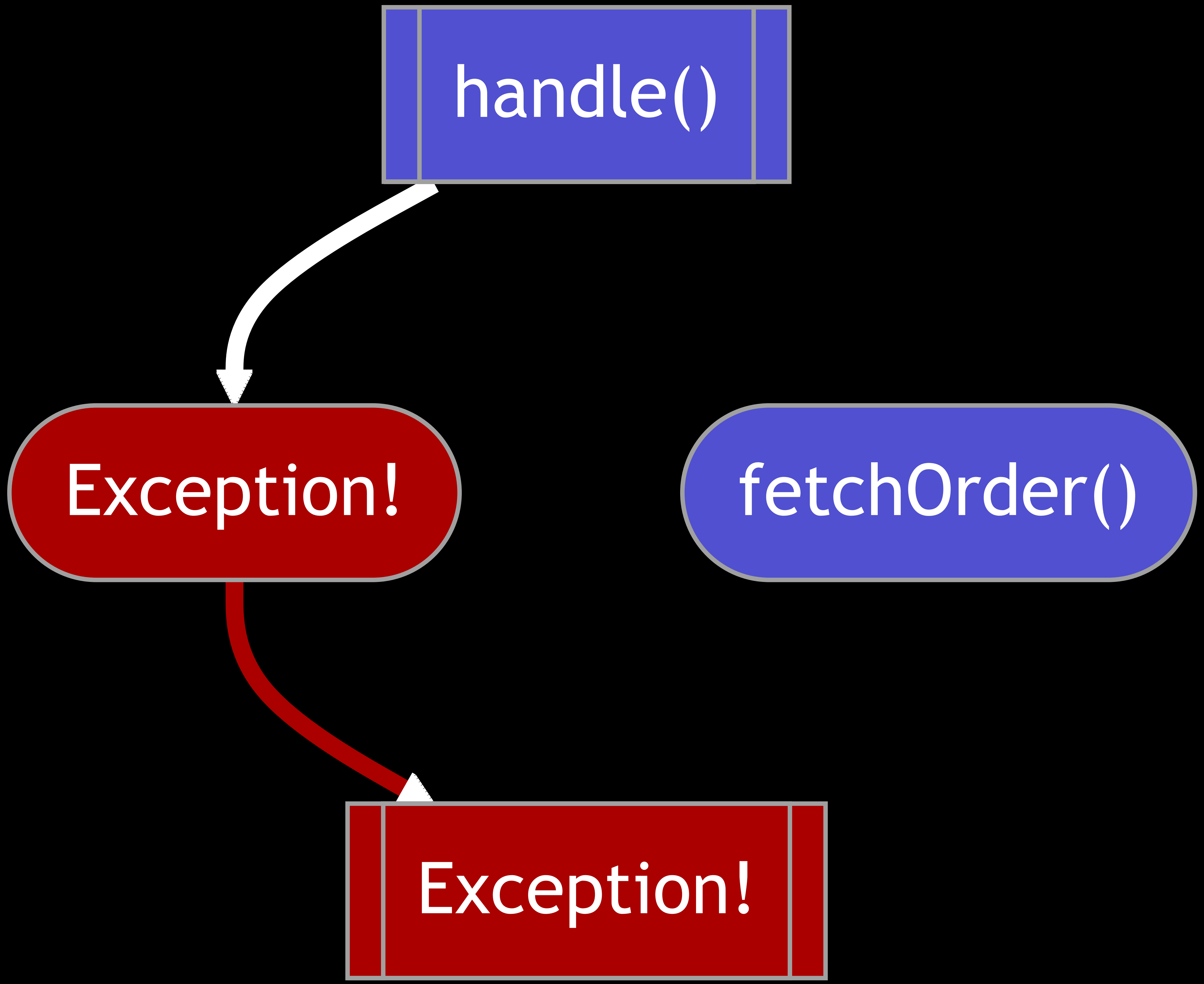
JAVA CONCURRENCY EXAMPLE

```
Response handle() {
  var user = executorService.submit(() -> findUser());
  var order = executorService.submit(() -> fetchOrder());
  var theUser = user.get(); // Join first thread
  var theOrder = order.get(); // Join second thread
  return new Response(theUser, theOrder);
}
```



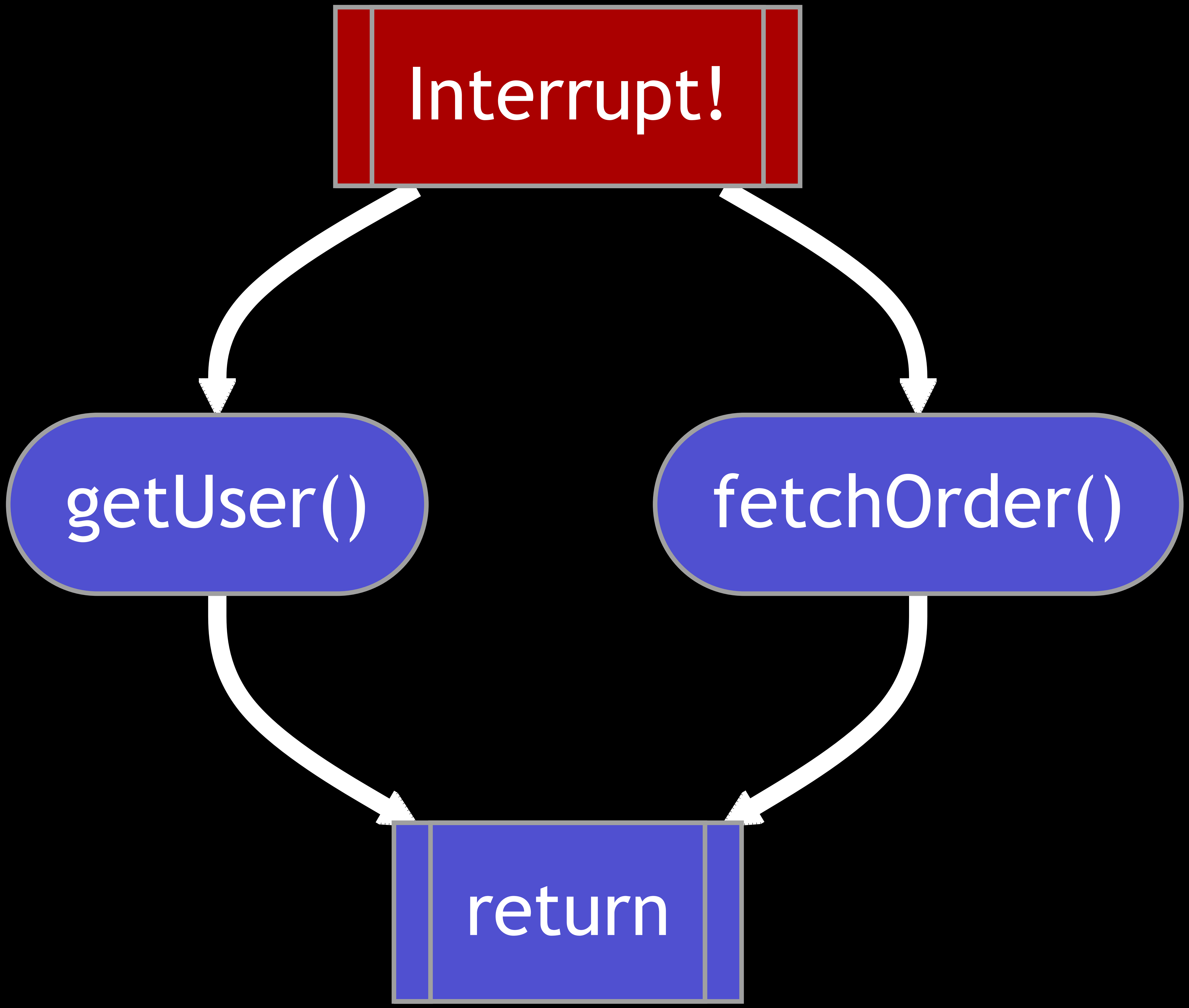
JAVA CONCURRENCY EXAMPLE

```
Response handle() {
  var user = executorService.submit(() -> findUser());
  var order = executorService.submit(() -> fetchOrder());
  var theUser = user.get(); // Join first thread
  var theOrder = order.get(); // Join second thread
  return new Response(theUser, theOrder);
}
```



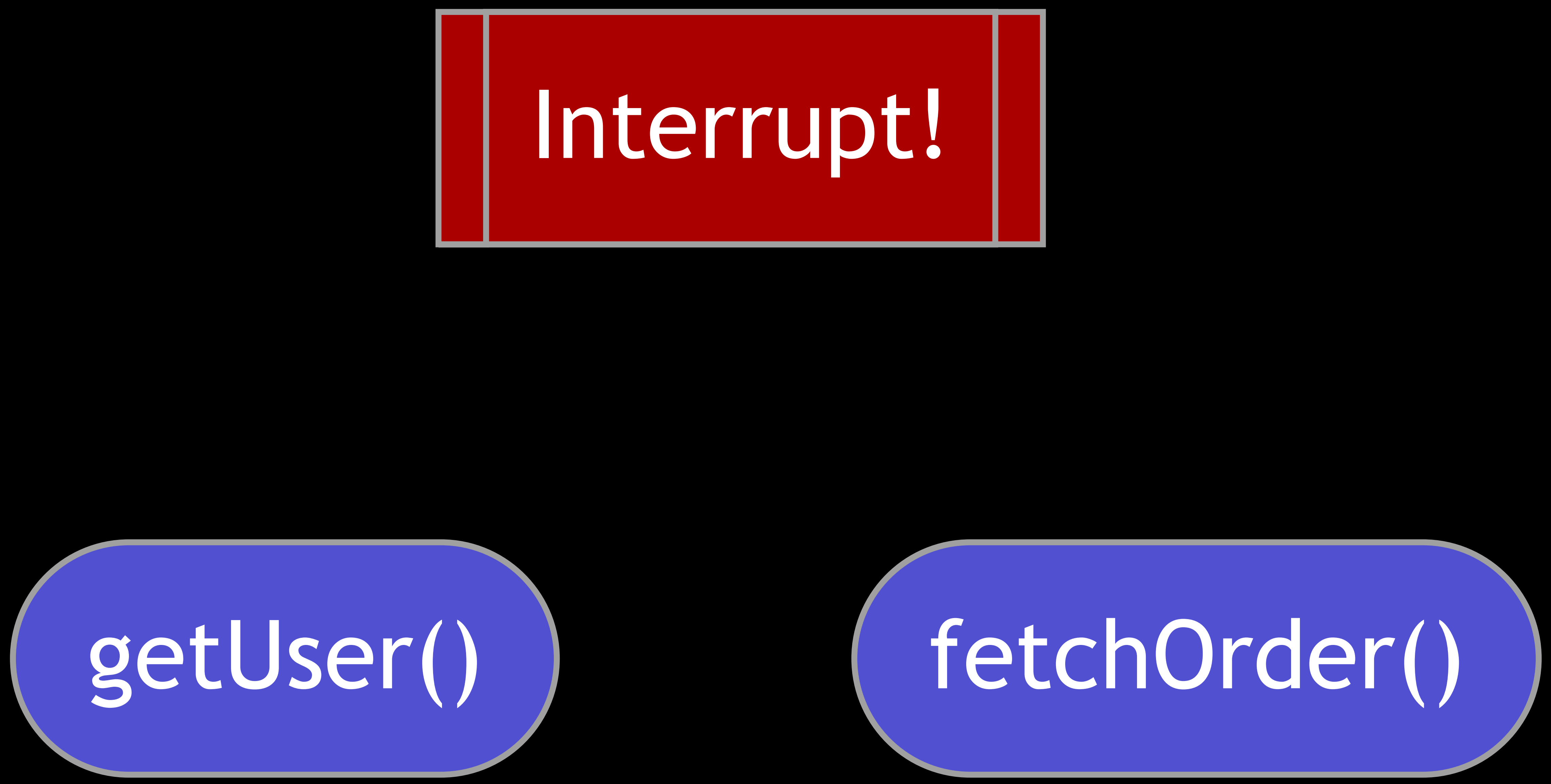
JAVA CONCURRENCY EXAMPLE

```
Response handle() {
  var user = executorService.submit(() -> findUser());
  var order = executorService.submit(() -> fetchOrder());
  var theUser = user.get(); // Join first thread
  var theOrder = order.get(); // Join second thread
  return new Response(theUser, theOrder);
}
```



JAVA CONCURRENCY EXAMPLE

```
Response handle() {
  var user = executorService.submit(() -> findUser());
  var order = executorService.submit(() -> fetchOrder());
  var theUser = user.get(); // Join first thread
  var theOrder = order.get(); // Join second thread
  return new Response(theUser, theOrder);
}
```



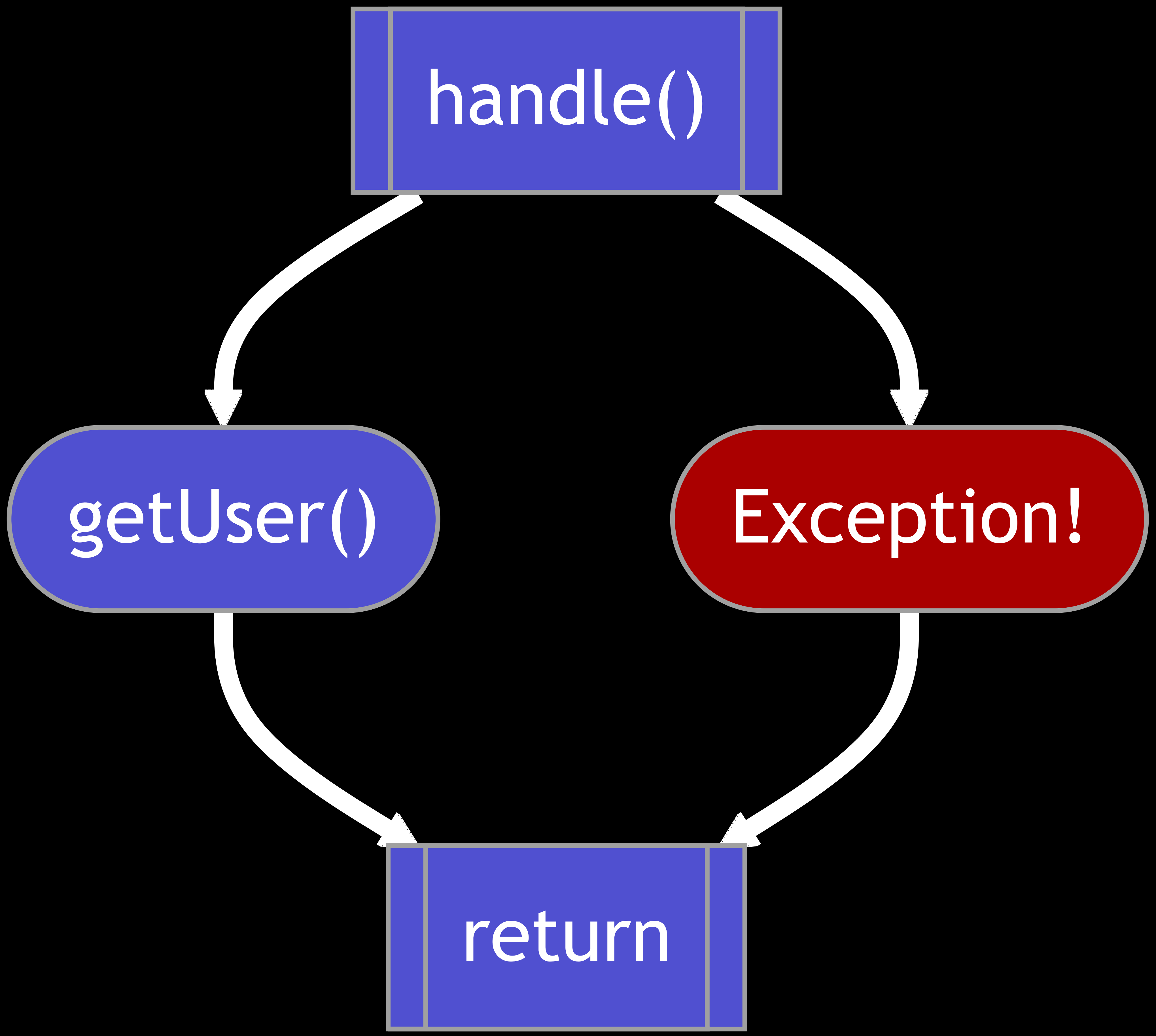
Interrupt!

getUser()

fetchOrder()

JAVA CONCURRENCY EXAMPLE

```
Response handle() {
  var user = executorService.submit(() -> findUser());
  var order = executorService.submit(() -> fetchOrder());
  var theUser = user.get(); // Join first thread
  var theOrder = order.get(); // Join second thread
  return new Response(theUser, theOrder);
}
```

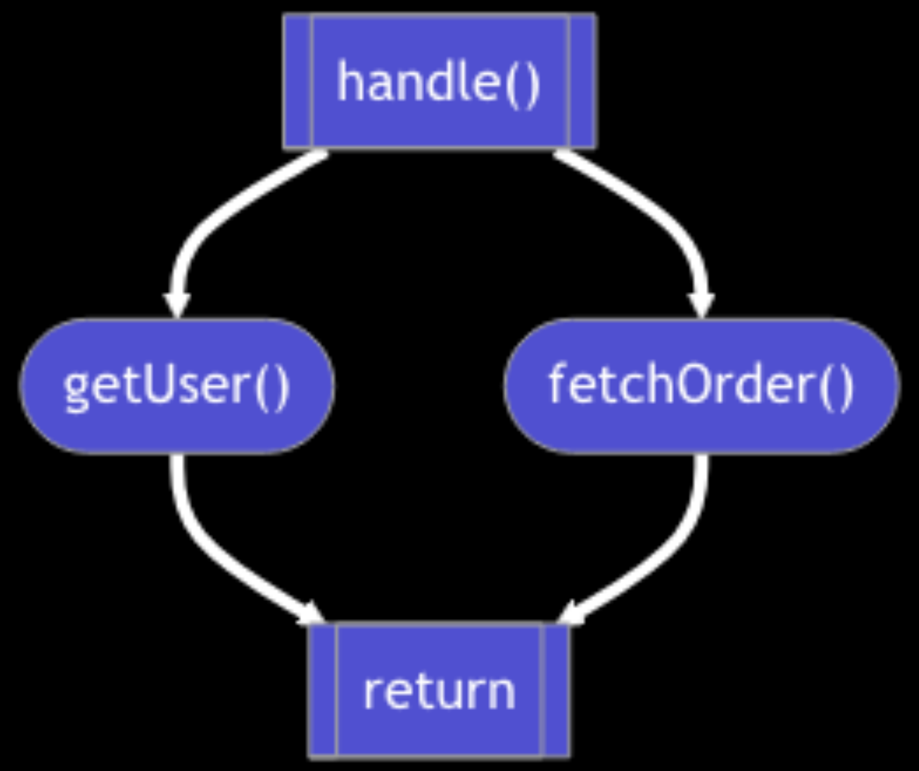


CONCURRENCY GIVES US DIFFICULT PROBLEMS

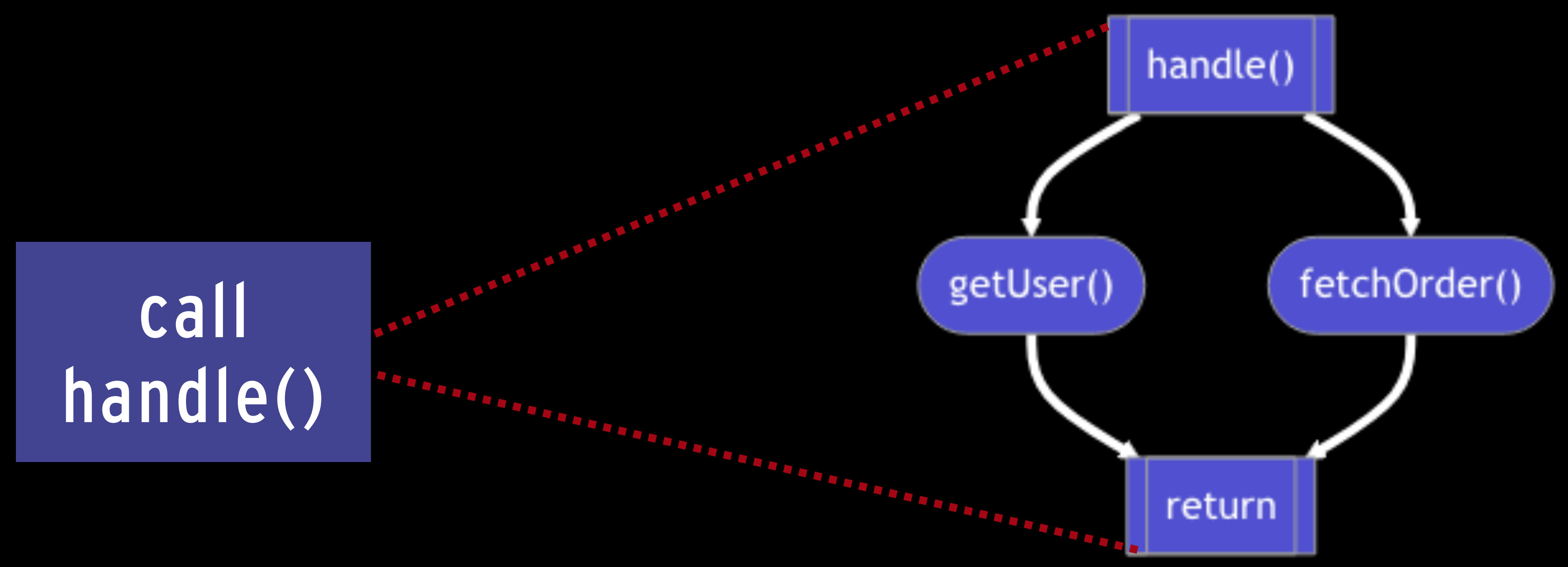
- We have multiple problems:
  - *Cancelling*: when a parent thread dies, the children are not cancelled.
  - *Error handling*: who should a thread report to if the parent thread is gone?
  - *Monitoring*: there is nothing in the runtime environment indicating a relationship between these threads.

- Tricky problems, and a solution often obscures the real intention of the code.

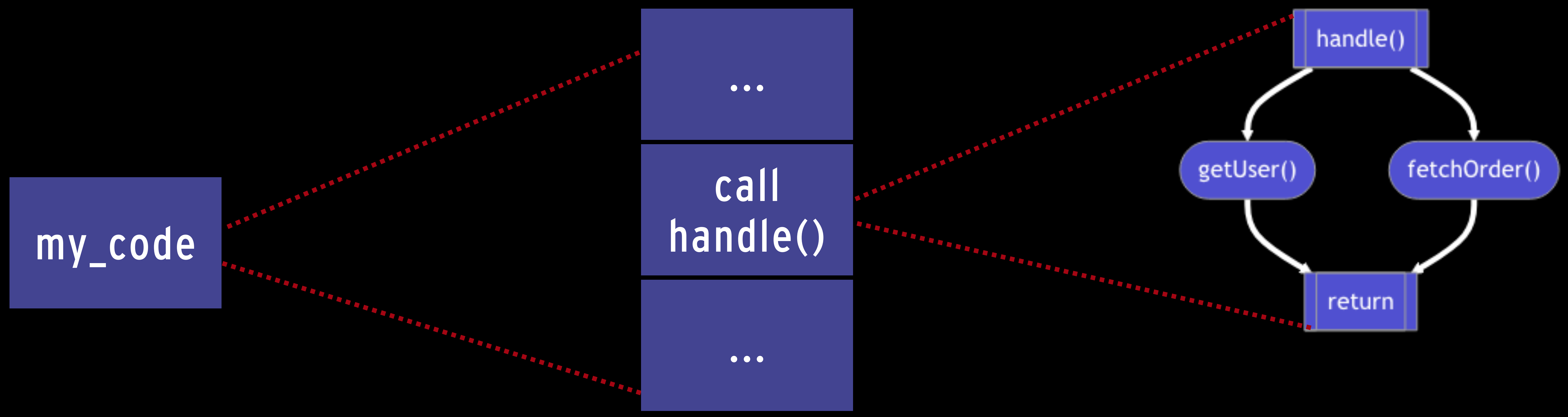
OUTSIDE PERSPECTIVE: WHAT THREADS DO MY FUNCTION CALLS LEAVE BEHIND?



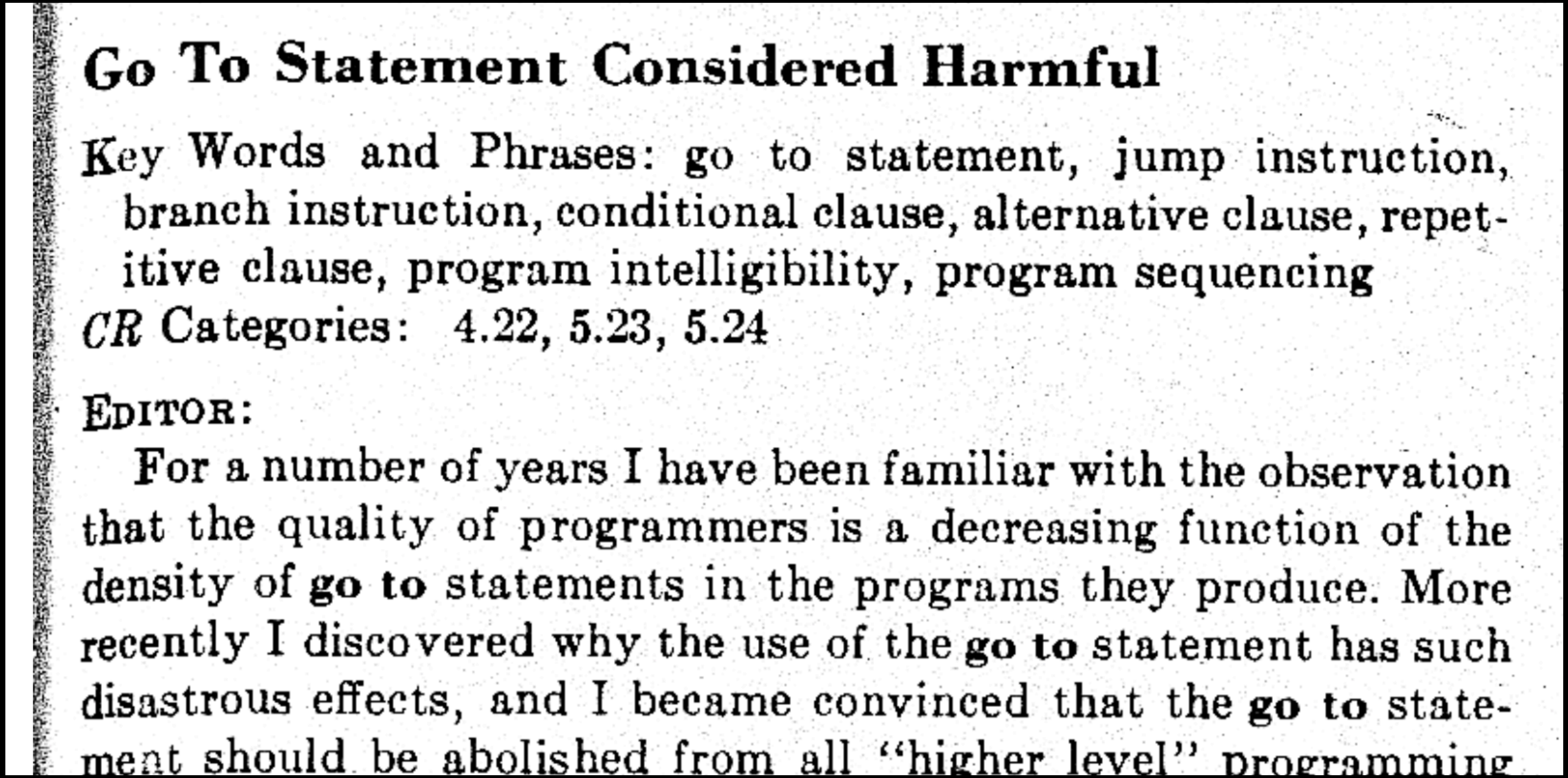
OUTSIDE PERSPECTIVE: WHAT THREADS DO MY FUNCTION CALLS LEAVE BEHIND?



OUTSIDE PERSPECTIVE: WHAT THREADS DO MY FUNCTION CALLS LEAVE BEHIND?



HISTORIC BACK REFERENCE



Go To Statement Considered Harmful

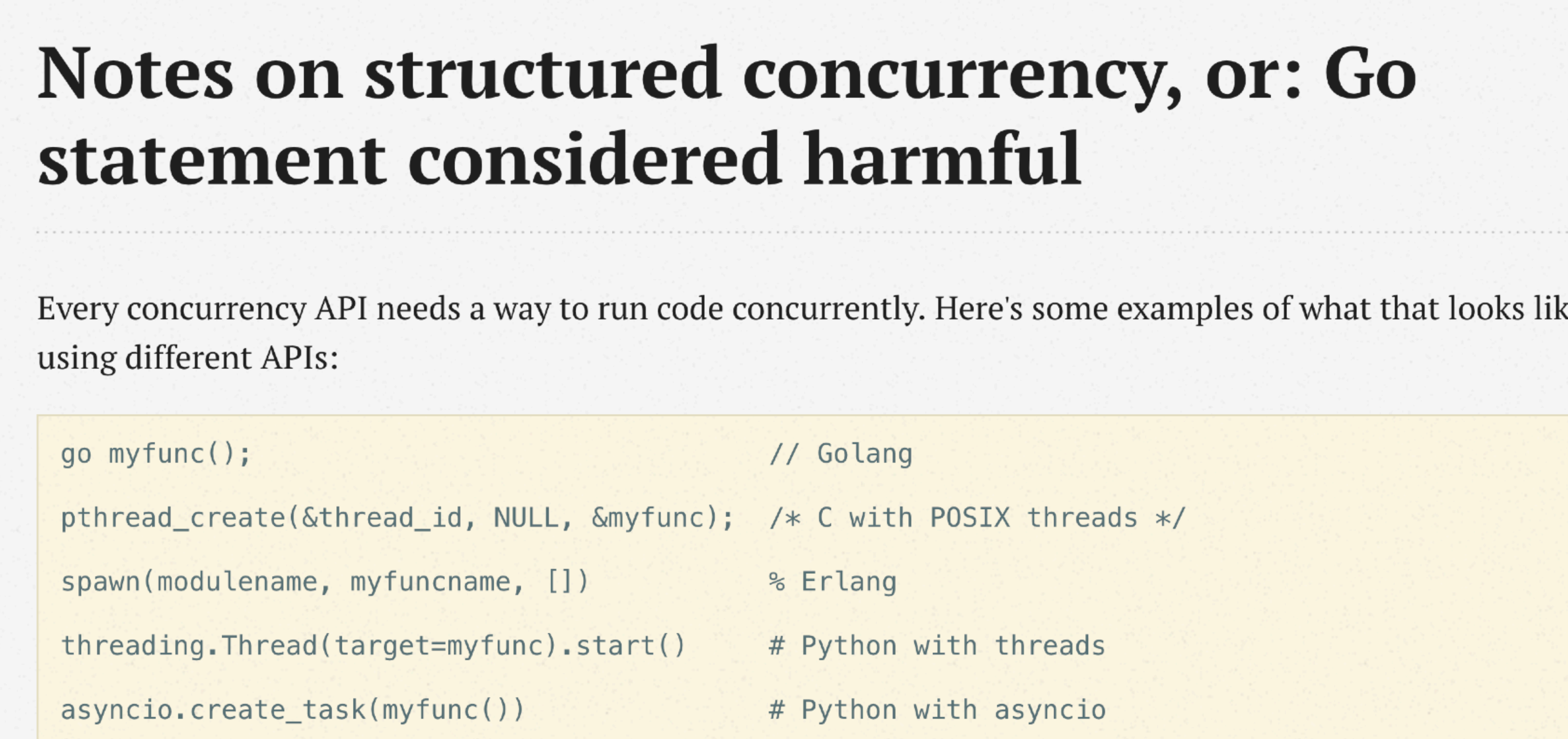
Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming

DIJKSTRA 1968



Notes on structured concurrency, or: Go statement considered harmful

Every concurrency API needs a way to run code concurrently. Here's some examples of what that looks lik using different APIs:

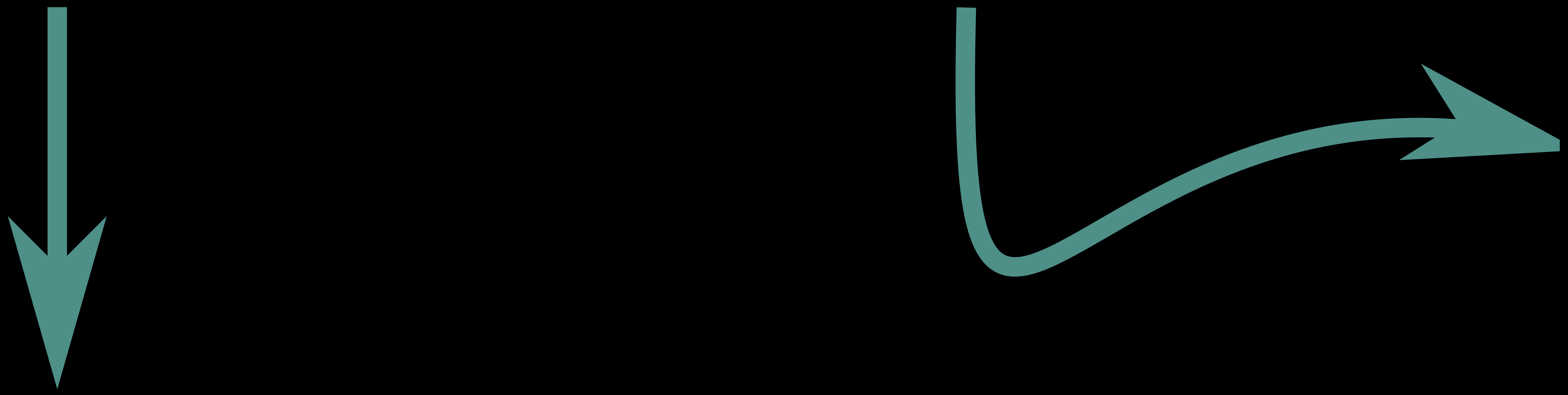
```
go myfunc();                                // Golang

pthread_create(&thread_id, NULL, &myfunc);  /* C with POSIX threads */

spawn(modulename, myfuncname, [])           % Erlang

threading.Thread(target=myfunc).start()     # Python with threads

asyncio.create_task(myfunc())               # Python with asyncio
```

SMITH 2018

GOTO DESTROYS LOCAL ANALYSIS

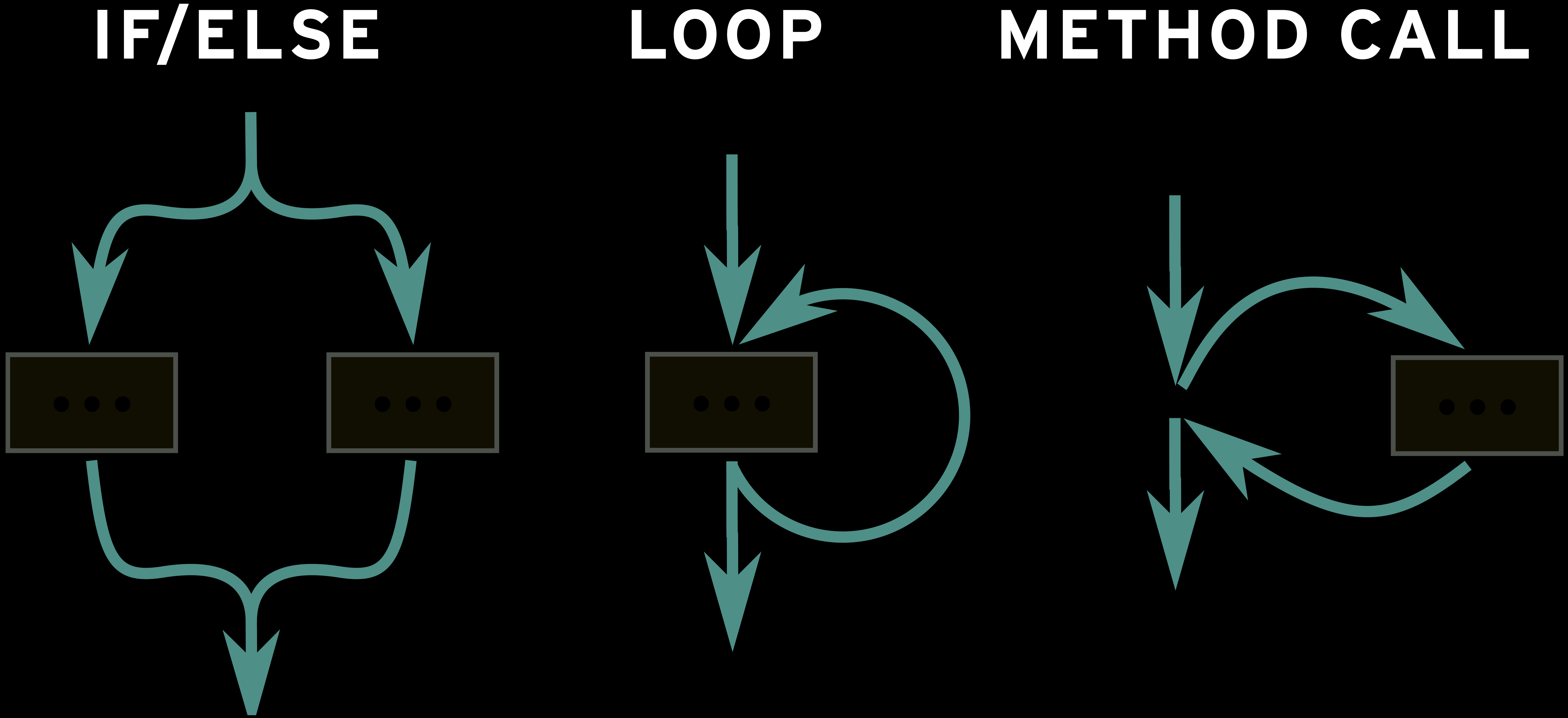
SEQUENTIAL CODE

GOTO



https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-co

STRUCTURED PROGRAMMING REQUIRES A SINGLE EXIT POINT



IF/ELSE

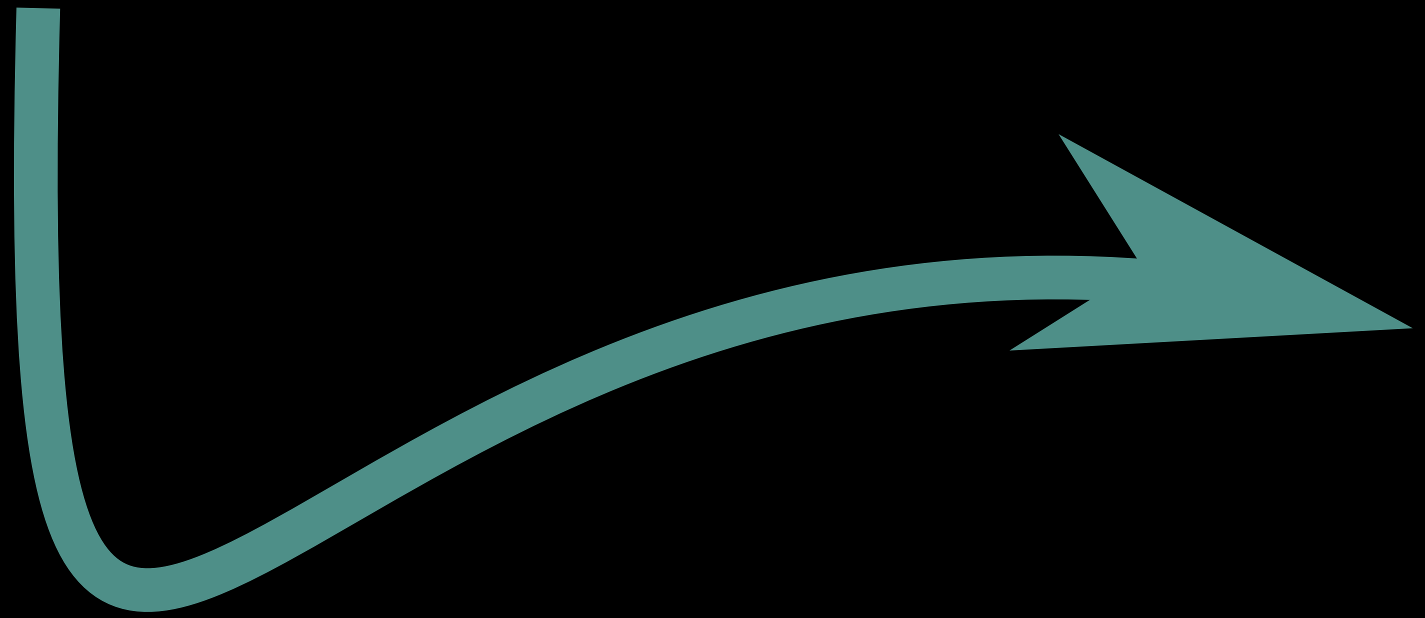
LOOP

METHOD CALL

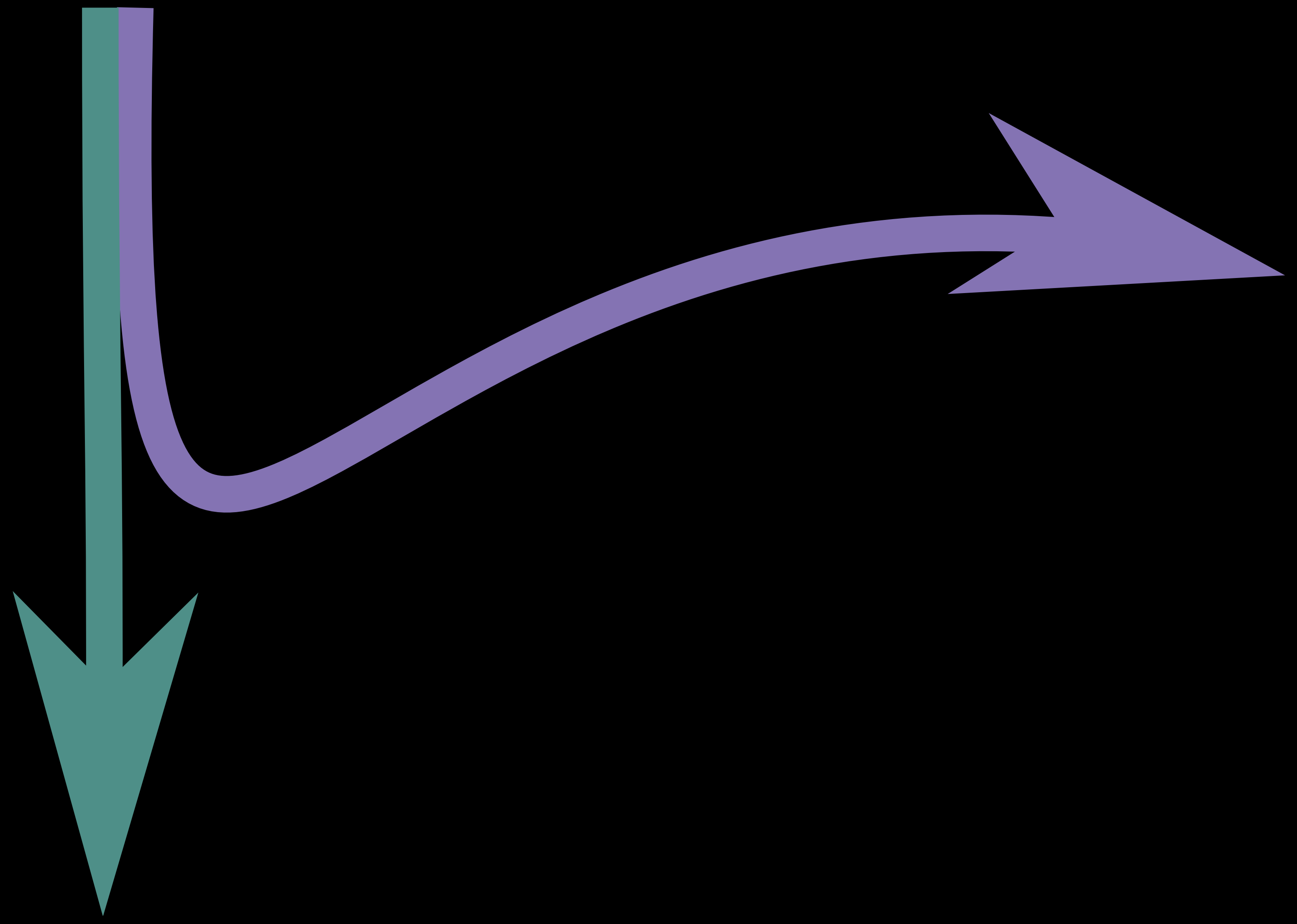
https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-con

SPAWNING OF NEW THREADS RESEMBLES GOTO

GOTO

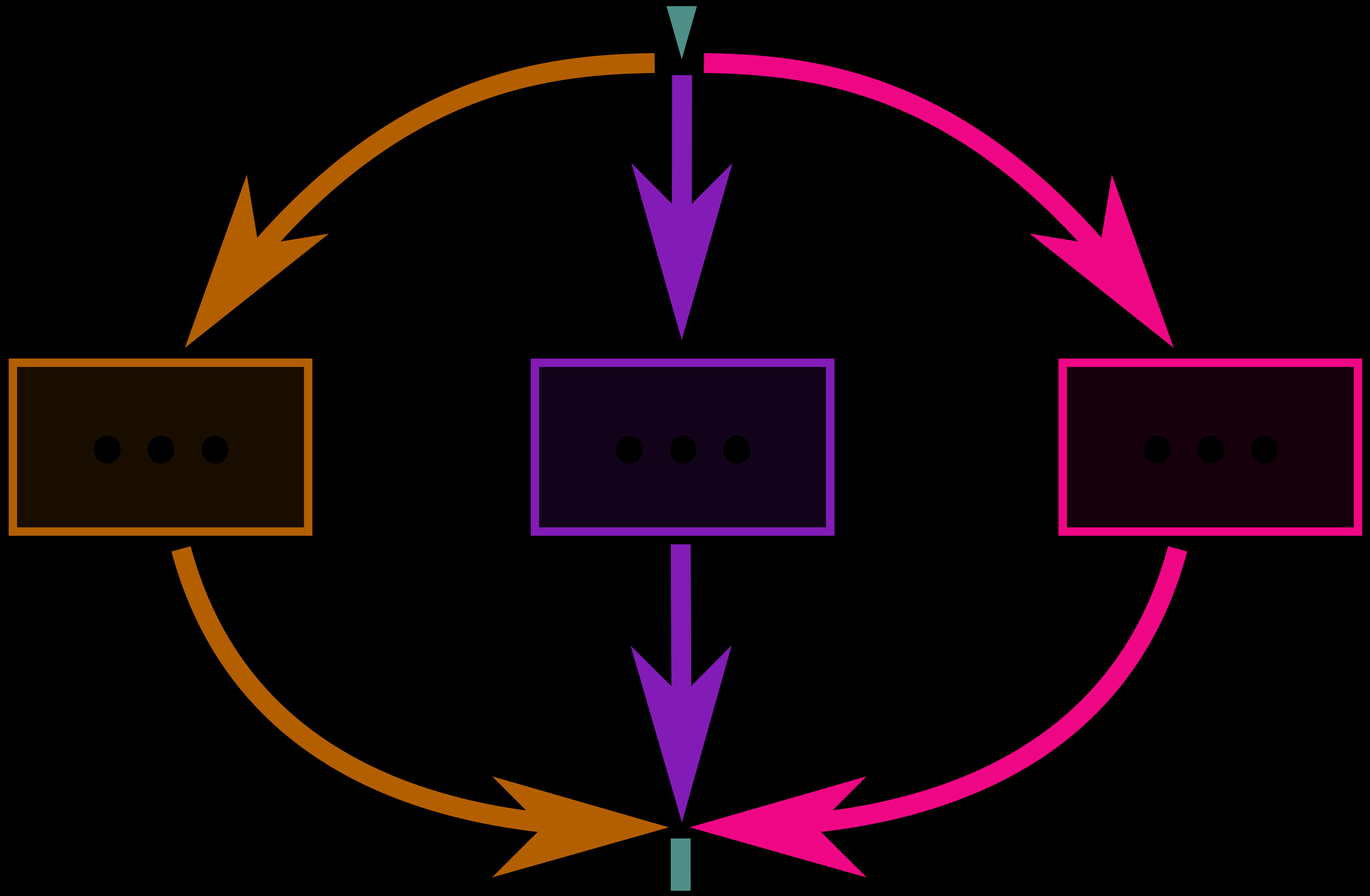


NEW THREAD



https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-con

CONCURRENCY SHOULD BE LIKE STRUCTURED PROGRAMMING



CALLISTA

https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-con

YET ANOTHER PERSPECTIVE: ADAPTING TO OUR STRENGTHS

The brain is good at reasoning about static structures but bad at parallel processes, so let's make the processes follow the structure of the code.

STRUCTURED CONCURRENCY: SCOPE OBJECT

- Structured concurrency introduces a scope object:
  - all threads started through the scope
  - the scope outlives all its child threads
  - the scope takes care of cancellation
  - the scope leaves no threads behind in case of exceptions
  - scopes can be nested
- We get:
  - Automatic resource management: never lose a thread
  - Automatic error handling: never lose an exception
  - A visible, hierarchical relation between all running threads

CONCURRENCY EXAMPLE REVISITED

```
Response handle() {
  var user = executorService.submit(() -> findUser());
  var order = executorService.submit(() -> fetchOrder());
  return new Response(user.get(), order.get());
}
```

```
Response handle() {
  try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    var user = scope.fork(() -> findUser());
    var order = scope.fork(() -> fetchOrder());
    scope.join();
    return new Response(user.get(), order.get());
  }
}
```

Java Structured Concurrency

```
suspend fun handle(): Response =
    coroutineScope {
        val user = async { findUser() }
        val order = async { fetchOrder() }
        Response(user.await(), order.await())
    }
```

Kotlin coroutines

DEMO

SCOPED VALUES

- We need to share data between our threads:
  - but we want to keep the hierarchical structure and the scope
  - we want to make sure that we control the life-cycle of data in our threads
- Kotlin uses a coroutine context: a map where parent threads can put values only visible to child threads
- Java gets something very similar with JEP 429 in Java 23: Scoped Values

JAVA: THREAD LOCAL VS. SCOPED VALUES

- Thread local variables have some issues:
  - No concept of scope
  - Mutable
  - Unbounded lifetime
  - Expensive inheritance
- Java scoped values are designed to work with structured concurrency:
  - The scope and lifetime of a value is clearly expressed in code structure
  - Immutable data is shared with callees and and child threads
  - Immutability makes resource sharing and thread creation much cheaper

SCOPED VALUE EXAMPLE

```
final static ScopedValue<String> CONTEXT = ScopedValue.newInstance();

private static void parent() {
    ScopedValue.where(CONTEXT, "myContext").run(() -> {
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            var result1 = scope.fork(() -> childComponent1());
            var result2 = scope.fork(() -> childComponent2());
            ...
        }
    });
}

static String childComponent1() throws InterruptedException {
    var context = CONTEXT.get();
    ...
}

static String childComponent2() throws InterruptedException {
    var context = CONTEXT.get();
    ...
}
```

CONCLUSION

- Structured concurrency libraries are now in many languages: C, Python, C#, Rust, Scala, Go, ...
- A few languages have it as part of the standard distribution: Kotlin, Swift and (soon) Java.
- *All* concurrency in a language could be structured concurrency - perhaps we will see new languages adopting this approach in the future?