

# Scoop about scopes

Rémi Forax

Université Gustave Eiffel - June 2023

# Rémi Forax ?

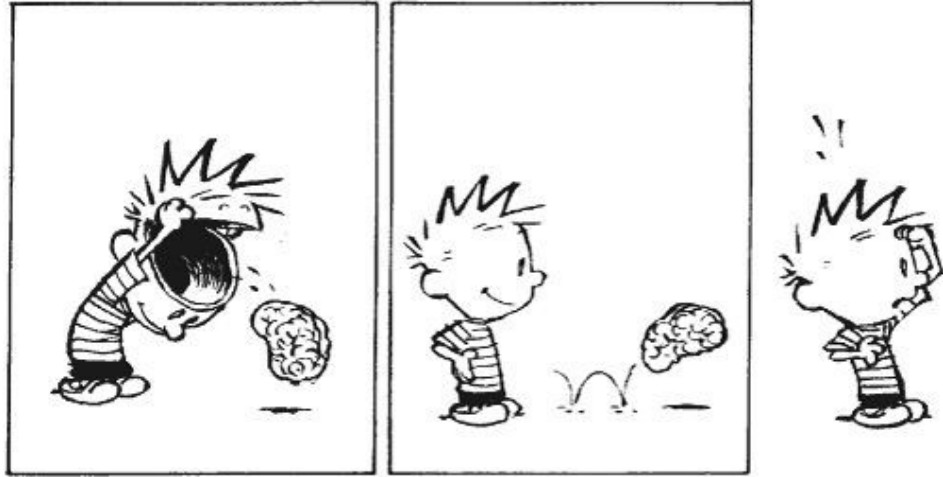
Tenured Assistant Prof at Gustave Eiffel University

Expert for Java spec

invokedynamic, lambda, module, text block, enhanced  
switch, record, sealed class, etc

OpenSource developer

ASM, [github.com/forax](https://github.com/forax)



CALVIN & HOBBS © BIL WATTERSON

Don't believe what I'm saying !

# Get an Episode of Rick & Morty

With a blocking call

```
static Episode getEpisode(int episodeId) throws IOException, InterruptedException {  
    try(var httpClient = HttpClient.newHttpClient()) {  
        var request = HttpRequest.newBuilder()  
            .uri(URI.create("https://rickandmortyapi.com/api/episode/" + episodeId))  
            .GET()  
            .build();  
  
        var response = httpClient.send(request, HttpResponse.BodyHandlers.ofInputStream());  
  
        var objectMapper = new ObjectMapper();  
        return objectMapper.readValue(response.body(), Episode.class);  
    }  
}
```

# Get an Episode of Rick & Morty


With a CompletableFuture

```
static CompletableFuture<Episode> getEpisode(int episodeId) {  
    try (var httpClient = HttpClient.newHttpClient()) {  
        var request = HttpRequest.newBuilder()  
            .uri(URI.create("https://rickandmortyapi.com/api/episode/" + episodeId))  
            .GET()  
            .build();  
        - return httpClient.sendAsync(request, HttpResponse.BodyHandlers.ofInputStream())  
            .thenCompose(response -> {  
                var objectMapper = new ObjectMapper();  
                Episode episode;  
                try {  
                    episode = objectMapper.readValue(response.body(), Episode.class);  
                } catch (IOException e) {  
                    return CompletableFuture.failedFuture(e);  
                }  
                return CompletableFuture.completedFuture(episode);  
            })  
    }  
}
```

# Async/Await

With an asynchronous call + async/await

```
static async Episode getEpisode(int episodeId) throws IOException, InterruptedException {  
    try(var httpClient = HttpClient.newHttpClient()) {  
        var request = HttpRequest.newBuilder()  
            .uri(URI.create("https://rickandmortyapi.com/api/episode/" + episodeId))  
            .GET()  
            .build();  
  
        var response = await httpClient.sendAsync(request, HttpResponse.BodyHandlers.ofInputStream());  
        var objectMapper = new ObjectMapper();  
        return objectMapper.readValue(response.body(), Episode.class);  
    }  
}
```



made up language: Java + JavaScript

# OpenJDK Project Loom

Users write synchronous code, the JDK executes asynchronous calls

## Virtual Threads

JDK Threads that can be attached/detached to/from an OS threads

- Preview in Java 19, Final in Java 21

# Virtual Threads

// platform threads

```
var pthread = new Thread(() -> {  
    System.out.println("platform " + Thread.currentThread());  
});  
pthread.start();  
pthread.join();
```

// virtual threads

```
var vthread = Thread.startVirtualThread(() -> {  
    System.out.println("virtual " + Thread.currentThread());  
});  
vthread.join();
```



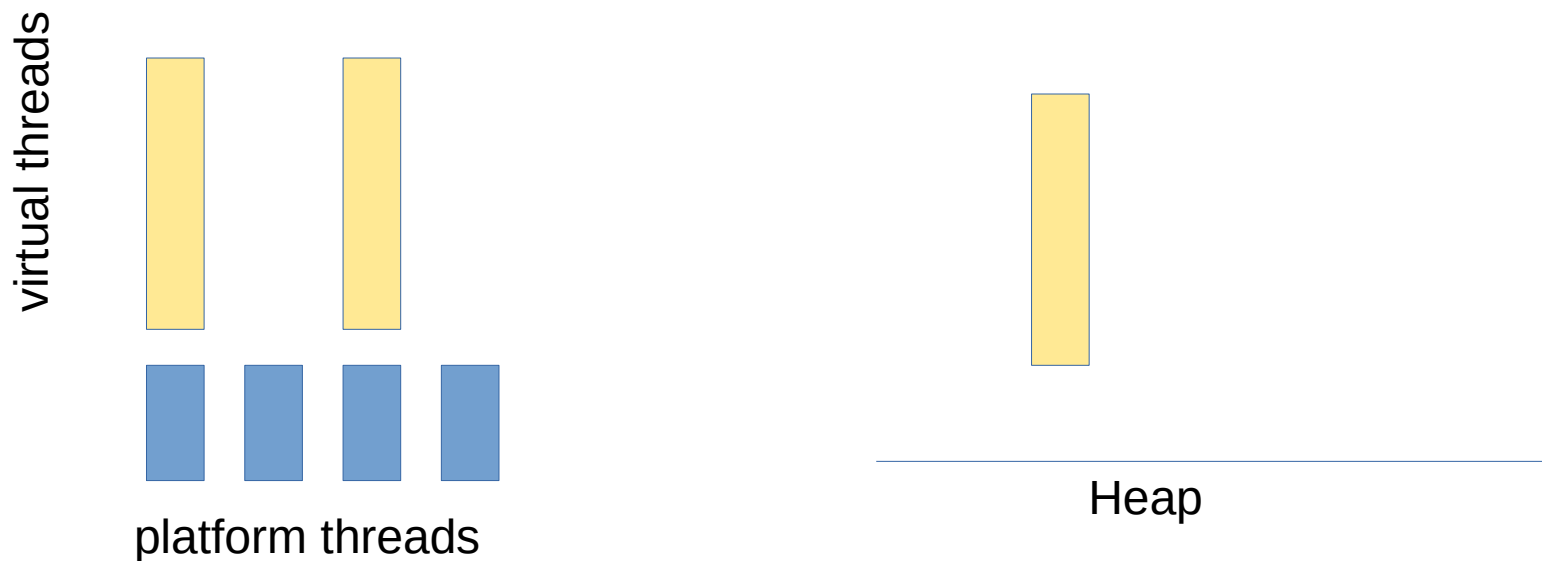
# Get an Episode of Rick & Morty

With an asynchronous call + virtual threads

```
static Episode getEpisode(int episodeId) throws IOException, InterruptedException {  
    try(var httpClient = HttpClient.newHttpClient()) {  
        var request = HttpRequest.newBuilder()  
            .uri(URI.create("https://rickandmortyapi.com/api/episode/" + episodeId))  
            .GET()  
            .build();  
  
        var response = httpClient.send(request, HttpResponse.BodyHandlers.ofInputStream());  
  
        var objectMapper = new ObjectMapper();  
        return objectMapper.readValue(response.body(), Episode.class);  
    }  
}
```

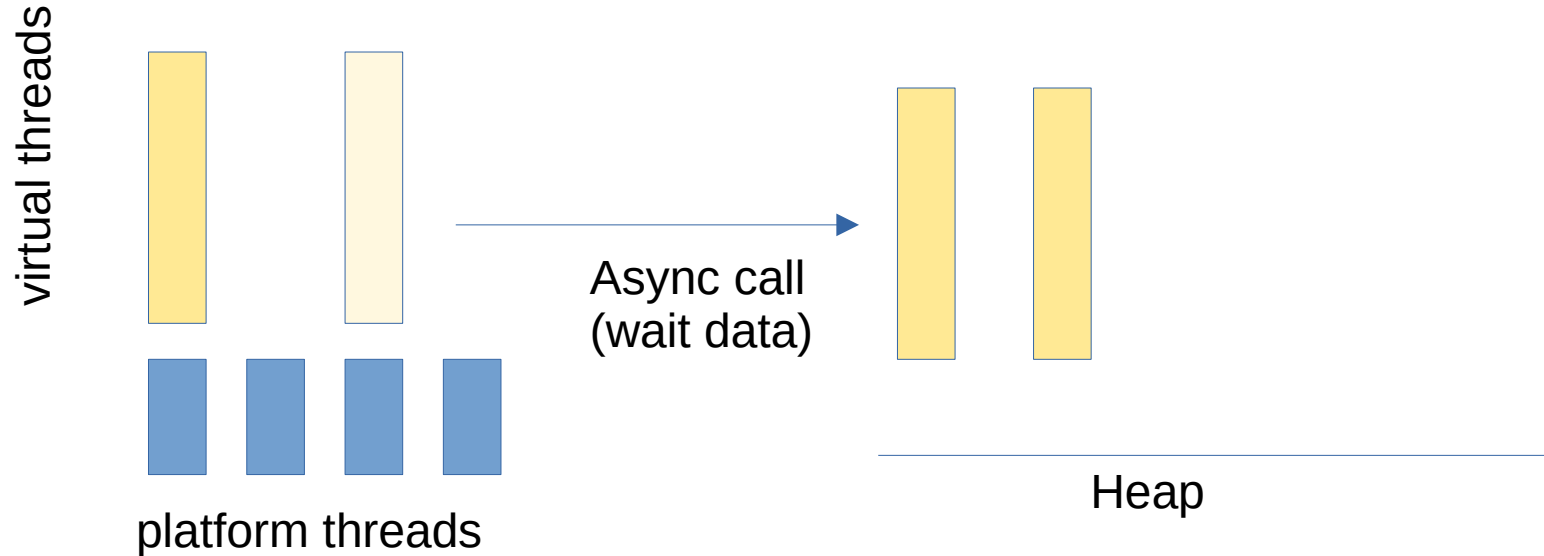
# Behind the scene (1/3)

Virtual threads run on top of platform (OS) threads



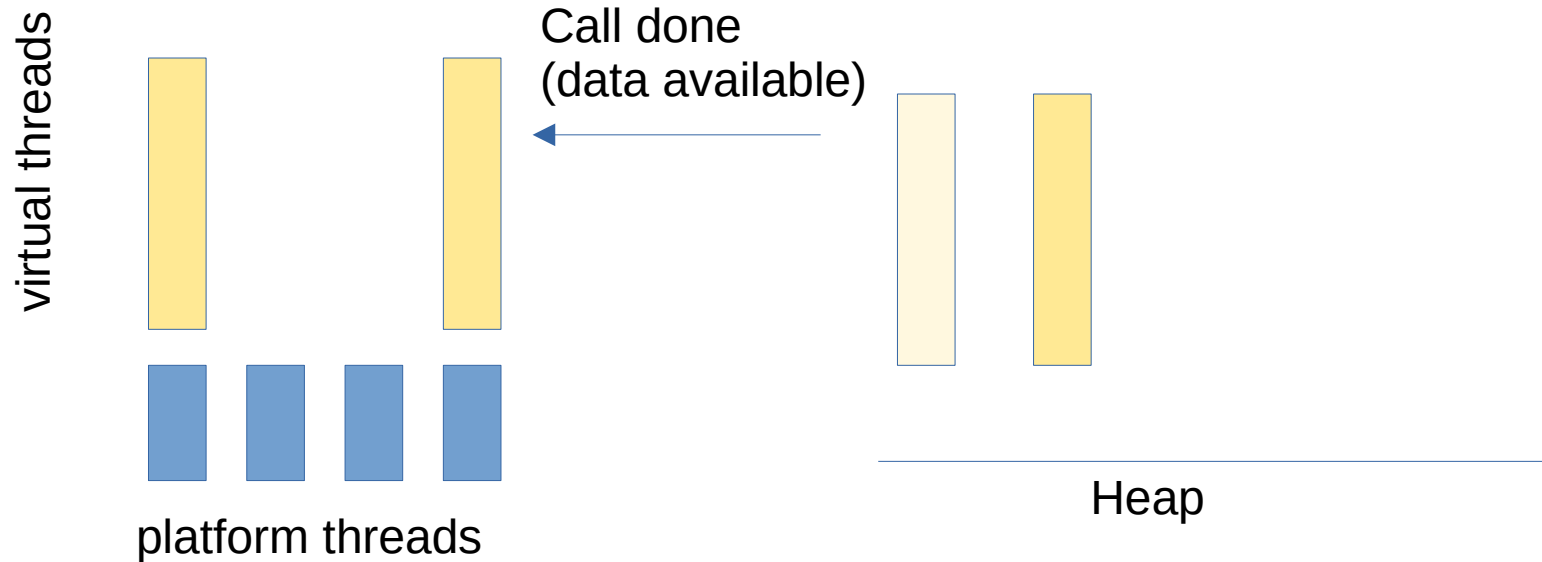
# Behind the scene (2/3)

Async call: the virtual thread is copied to the heap



# Behind the scene (3/3)

Data are available: the virtual threads is copied back on a stack



How to run several async calls  
in parallel ?

# ExecutorService API ?

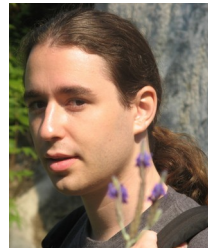
```
var executor = Executors.newVirtualThreadPerTaskExecutor();  
var future1 = executor.submit() -> {  
    Thread.sleep(10);  
    return ...  
});  
var future2 = executor.submit() -> {  
    Thread.sleep(1_000);  
    return ...  
});  
executor.shutdown();  
var result = future1.get() + future2.get();  
System.out.println(result);
```

← Oooooooooops

# Structured Concurrency

## Structured programming for concurrency

- Structured code (goto is harmful)
  - if, while, etc are better than goto
- `ExecutorService.submit()` is like a goto !



WED 25 APRIL 2018

## 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 like 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
```

There are lots of variations in the notation and terminology, but the semantics are the same: these all arrange for `myfunc` to start running concurrently to the rest of the program, and then return immediately so that the parent can do other things.

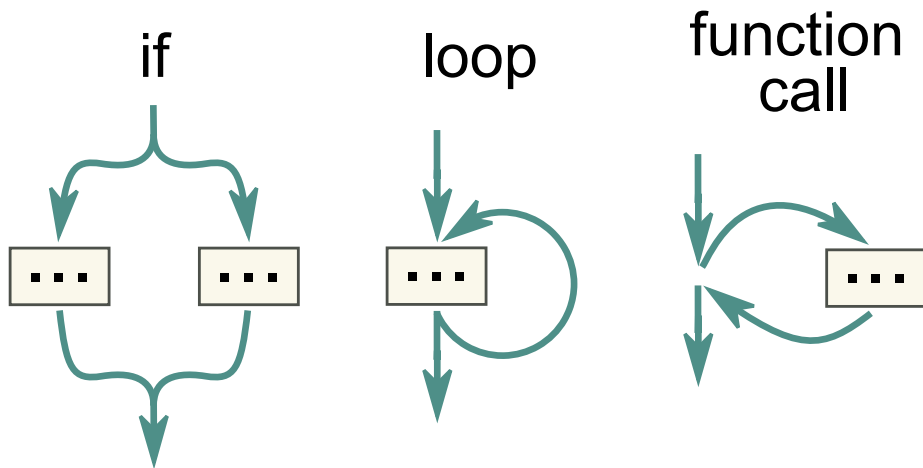
<https://vorp.us.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/>



# Structured Concurrency

## Structured programming for concurrency

- Structured code (goto is harmful)  
if, while, function calls, etc are better than goto



# Structured Concurrency

Is `ExecutorService.submit()/Future.get()` a goto ?

Change in Java 19

`ExecutorService` now implements `AutoClosable`

```
try(var executor = ...) {  
    var future = executor.submit(() -> { ... });  
    var future2 = executor.submit(() -> { ... });  
    ...  
} // all threads are dead here
```

# Structured Concurrency

Parent/child relationship between a task and its subtasks

- No runaway threads
  - all threads have finished at the end
- No ignored exceptions
  - exceptions are not be swept under the rug

# StructuredTaskScope

run several async calls in parallel

(preview feature in Java 21)

# StructuredTaskScope

```
try(var sts = new StructuredTaskScope<...>()) {  
    ... // starts threads  
    sts.join(); // wait for all threads  
    ... // collect results  
} // clean threads
```

# Example of StructuredTaskScope

```
try(var sts = new StructuredTaskScope<Integer>()) {  
    var task = sts.fork(() -> 3);  
    var task2 = sts.fork(() -> 42);  
    sts.join();  
    var result = task.get() + task2.get();  
}
```

# Big Picture

As a user, it's two levels of API

- Primary API (STS)
  - **try**(**var** sts = **new** STS<...>()) { ...
  - sts.join()
- Secondary API (Subtask)
  - **Subtask**<...> task = sts.fork(...)
  - task.state(), task.get() or task.exception()



SUCCESS, FAILED, UNAVAILABLE

# STS and STS subclasses

StructuredTaskScope offers two subclasses

- STS.ShutdownOnSuccess(), stop when one task succeed
- STS.ShutdownOnFailure(), stop if one task failed

StructuredTaskScope can also be inherited

```
class MySTS extends STS<Integer> {  
    // called concurrently by all threads after completion  
    public void handleComplete(Task<? extends Integer> task) {  
        task.state() // only SUCCESS or FAILED  
        ...  
    }  
}
```



Demo

# ShutdownOnSuccess

The result is available on the STS

```
int value;  
try(var sts = new STS.ShutdownOnSuccess<Integer>()) {  
    sts.fork(() -> ...);  
    sts.fork(() -> ...);  
    value = sts.join()  
        .result(); // T or throws ExecutionException  
}
```

# ShutdownOnFailure

The exception is available on the STS

```
int result;  
try(var sts = new STS.ShutdownOnFailure()) {  
    var task1 = sts.fork(() -> ...);  
    var task2 = sts.fork(() -> ...)  
    sts.join()  
    .throwIfFailed(); // may throw ExecutionException  
    result = task1.get() + task2.get();  
}
```

# Timeout

```
try (var scope = new StructuredTaskScope<>()) {  
    var task1 = scope.fork(() → ...);  
    var task2 = scope.fork(() -> ...);  
  
    try {  
        scope.joinUntil(Instant.now().plus(Duration.ofMillis(100)));  
    } catch (TimeoutException e) {  
        ...  
    }  
  
    System.out.println(task1.state()); // may be UNAVAILABLE if timeout  
    System.out.println(task2.state()); // may be UNAVAILABLE if timeout  
}
```

Iteratively improve the API  
(for Java 22 ??)

# STS API Issues for me

Issues I would like to fix:

- ShutdownOnXXX can be misused if `throwIfFailed()/result()` are not used
- Exceptions are erased to Throwable and wrapped
- `STS.handleComplete(Subtask<...>)` is too dangerous !
  - Also SubTask states are different inside/outside of `handleComplete()`

Demo

# STSShutdownOnSuccess

fork() returns void, the exception is propagated by joinAll()

```
try (var scope =  
    new STSShutdownOnSuccess<Integer, RuntimeException>()) {  
    scope.fork() -> ...;  
    scope.fork() -> ...;  
    var result = scope.joinAll(); // Integer or throws RuntimeException  
    System.out.println(result);  
}
```



# STSShutdownOnFailure

Suppliers are typed by the return values, STS by the exception

```
try (var scope =  
    new STSShutdownOnFailure<IOException>()) {  
    Supplier<Integer> supplier1 = scope.fork(...);  
    Supplier<Integer> supplier2 = scope.fork(...);  
    scope.joinAll(); // may throw IOException  
    System.out.println(supplier1.get() + supplier2.get());  
}
```

# STSAStream

joinAll() provides a stream of the finished tasks (Result)

```
try (var scope = new STSAStream<Integer, IOException>()) {  
    SubTask<Integer,IOException> task1 = scope.fork(...);  
    SubTask<Integer,IOException> task2 = scope.fork(...);  
  
    List<Result<Integer,IOException>> list =  
        scope.joinAll(stream -> stream.toList());  
    System.out.println(list);  
}
```

Result<T,E> acts as an union: Success(T) | Failed(E)

# STSAStream short circuit

If the stream finished, invocables still running are cancelled

```
try (var scope = new STSAStream<Integer, IOException>()) {  
    scope.fork(...);  
    scope.fork(...);  
  
    Optional<Integer> optional =  
        scope.joinAll(s -> s.flatMap(Result::keepOnlySuccess).findFirst());  
  
    System.out.println(optional);  
}
```

# Executive Summary

# Summary

Structured concurrency idea is cool :)

- JEP 453: Structured Concurrency (Java 21)
  - <https://openjdk.org/jeps/453>
- and in the future (maybe?)  
<https://github.com/forax/loom-fiber/tree/java21>