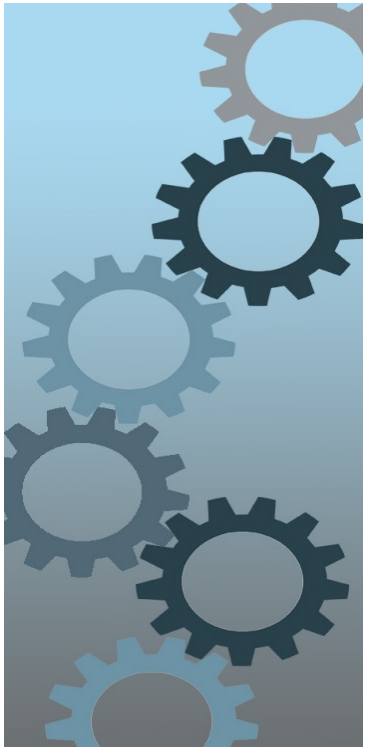
HSLU Hochschule Luzern

Programming Concepts And Paradigms

Loom
Structured Concurrency



Marcel Baumann

Content

- Examples of new Idioms
  - Auto Closeable *java.lang.AutoCloseable*
  - *Instanceof* and pattern matching
  - *Text blocks*
- Project Loom

AutoCloseable

- Finalizers are obsolete (*JEP 421 deprecate Finalization for Removal in JDK 18)*
  - They are a design error because they cannot be implemented efficiently with garbage collectors
- AutoCloseable and try with resources *solves the problem!*
  - Compatibility by updating Closeable
    Closeable implements AutoCloseable
  - Compiler eliminates verbosity and programmer’s errors
  - Mix API and language extension

Modern equals() - instanceof

```
class T {

   ...

   @Override
   public boolean equals(Object obj) {
       return (obj instanceof T that) &&
           (Objects.equals(x, that.x) && ...;
```

Modern toString() - text block

```
class T {

   ...

   @Override
   public String toString() {
        return """
            Activity[date=%s, code=%s]
            """.formatted(date(), code());
   }
```

Control Questions

1. Why are idioms relevant for a programming language?
2. Give an example of an idiom in your favorite programming language
3. When is the close() function of an *auto closeable* called?
4. Is the instanceof construct useful? Why?
5. How can you discourage developer to use a class or a method in Java?

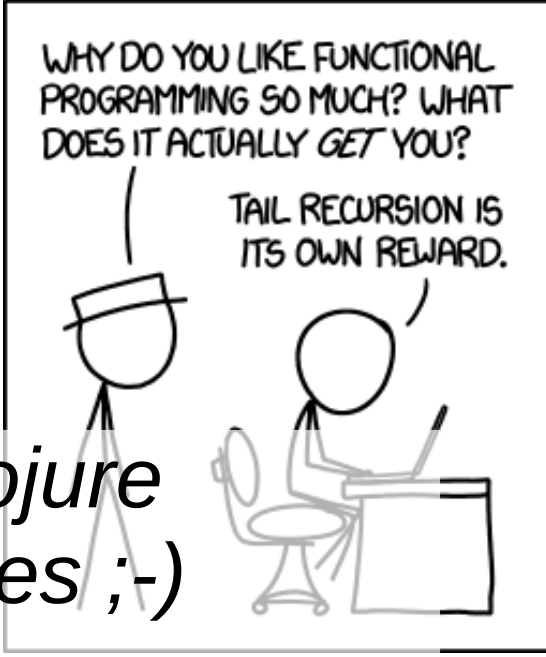
Loom

The green threads concept in the first version of Java is back. And improved to avoid blocking the underlying OS thread.

Virtual threads are an answer to the costs of asynchronous design

- Virtual Threads (JEP 425, 436, 429)
- Delimited Continuations
- Structured Concurrency (JEP 428, 437)
- Scoped Values (JEP 429, JEP 446)

Loom



- *All three features are available in Clojure (also in Scheme and Lisp) for decades ;-)*
- *Virtual threads exist in Smalltalk, Erlang, Go, etc.*

Virtual Threads

Virtual threads (Fiber) are just threads that are scheduled by the Java virtual machine rather than the operating system

- Normal Priority
- Daemon Threads
- No permissions with Security Manager
- Inactive threads in a thread group
- No support for Thread suspend, resume and stop APIs.

Delimited Continuations

- A Continuation (similar to coroutine) is a stack of function calls
- It can be stopped and stored in the heap at some point (with yield) and restarted afterward (with run)
- A continuation has no scheduler, so you have to write your own

Tail-Call Elimination

A recursive function is tail recursive when a recursive call is the last thing executed by the function.

Tail-call elimination is a compile-level optimization that is aimed to avoid stack overflow when calling a recursive method.

Tail-call elimination support in Java is still a future work.

Tail-call elimination is the death of loops.

Loom

```
Thread thread = Thread.startVirtualThread(runnable);

ExecutorService executor = Executors.newVirtualThreadExector();


executor.submit(runnable);
executor.submit(callable);
```

History of Concurrency (1/3)

- Processes
  - Create an operating system *heavy* process
- Threads
  - Create an operating system *light* thread
- Hand coded synchronization
  - Synchronized, volatile

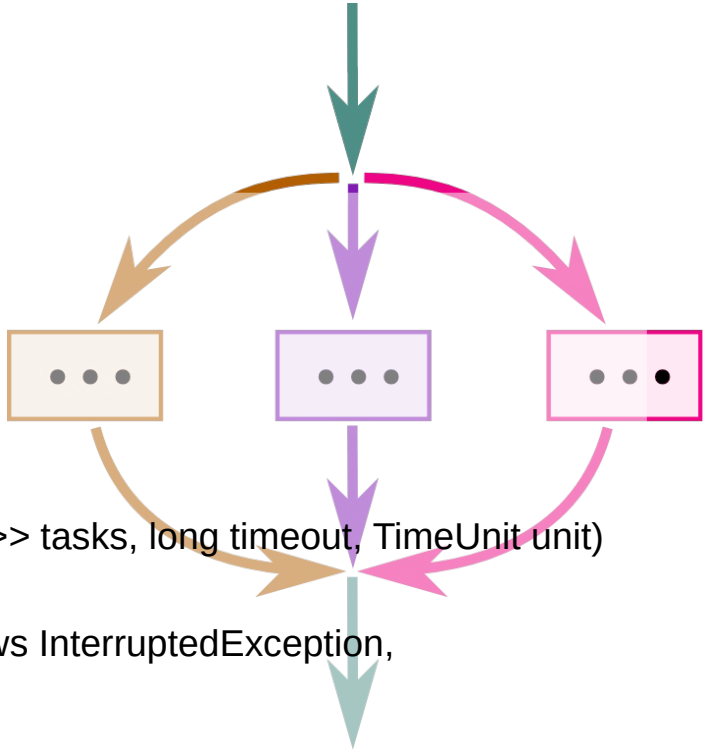
History of Concurrency (2/3)

- Executors
  - Simple creation of a thread
- Fork / Join (Java 7) → *too complicated to implement* ***divide and conquer***
- Concurrency Library
  - Concurrent collections
  - Future

History of Concurrency (3/3)

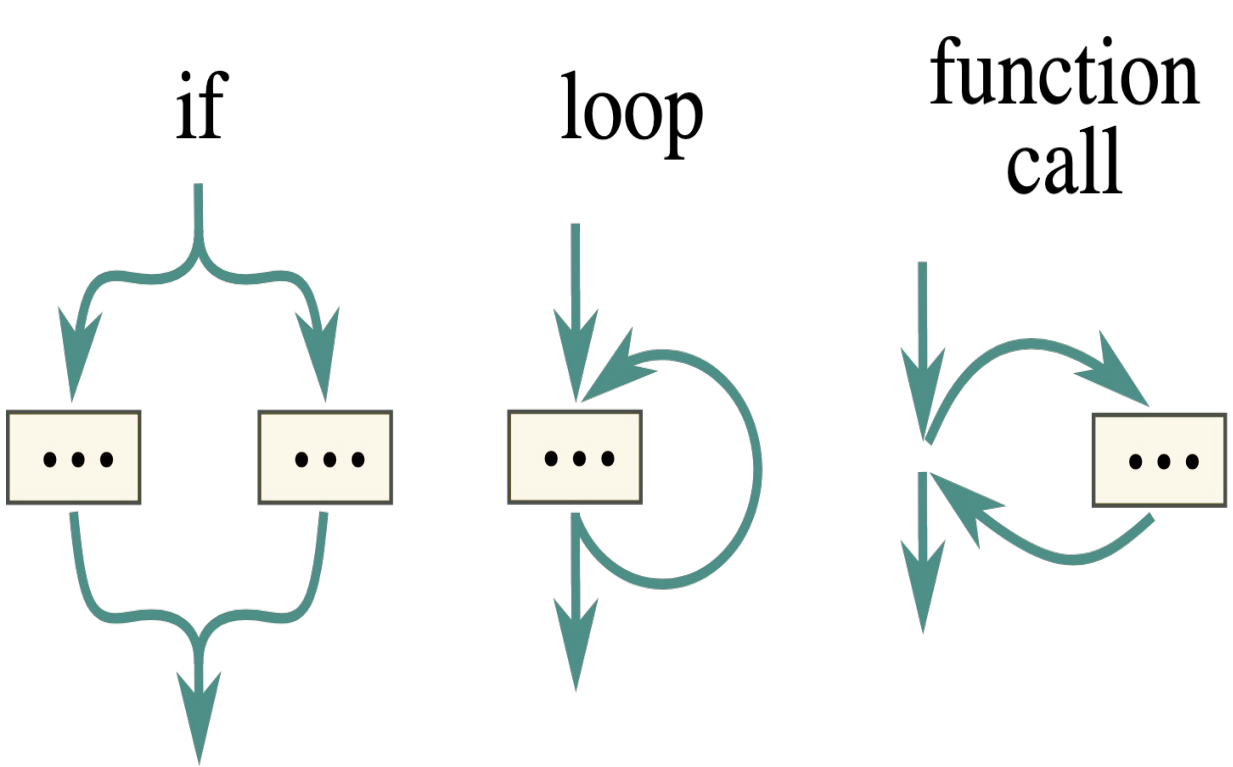
- Asynchronous Model
  works only if every block is asynchronous!
  - Completable Future
    - Supply Async()
- Stream and Composition
  - thenCompose(…), thenCombine(…)
  - thenApply(…)
  - AllOff(…)

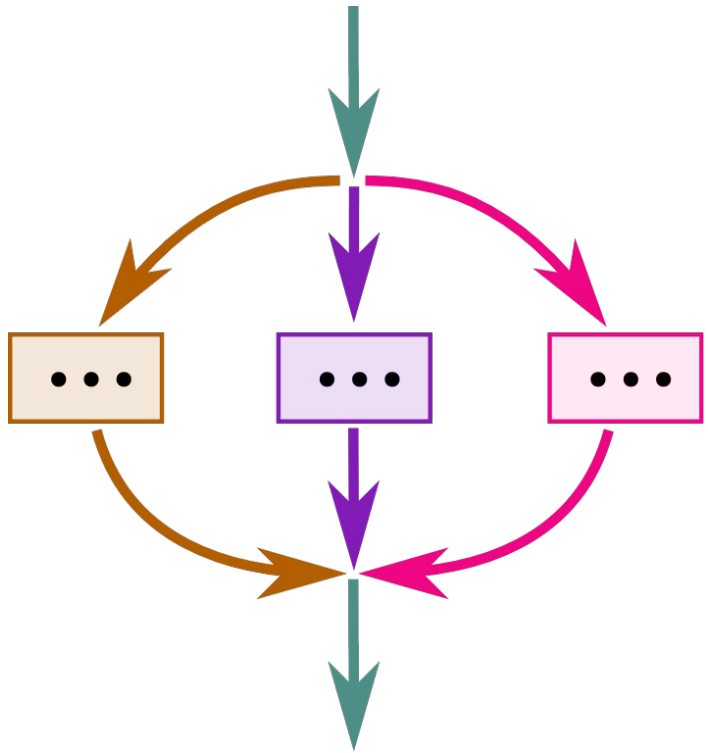
Structured Concurrency



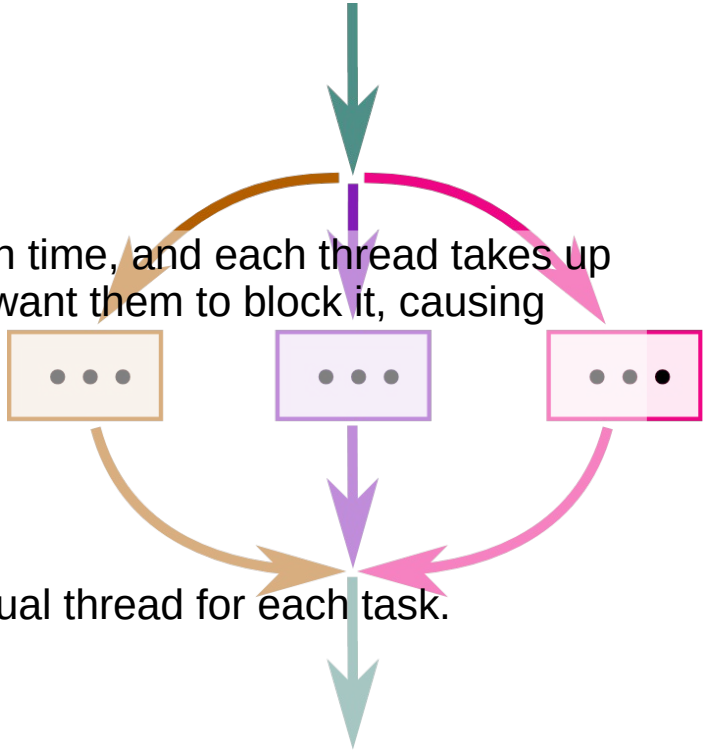
- Synchronization Tools
  - Fork And Join *without boilerplate*
  - Future
  - ExecutorService
    - <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit) throws InterruptedException
    - <T> T invokeAny(Collection<? extends Callable<T>> tasks) throws InterruptedException, ExecutionException
  - CompletableFuture
- The goal is
  - No complex synchronization to wait for completion
  - Follow the flow of the program, do not create a new flow (see for example async for a new flow with rules such as only async calls)

Structured Concurrency Idea





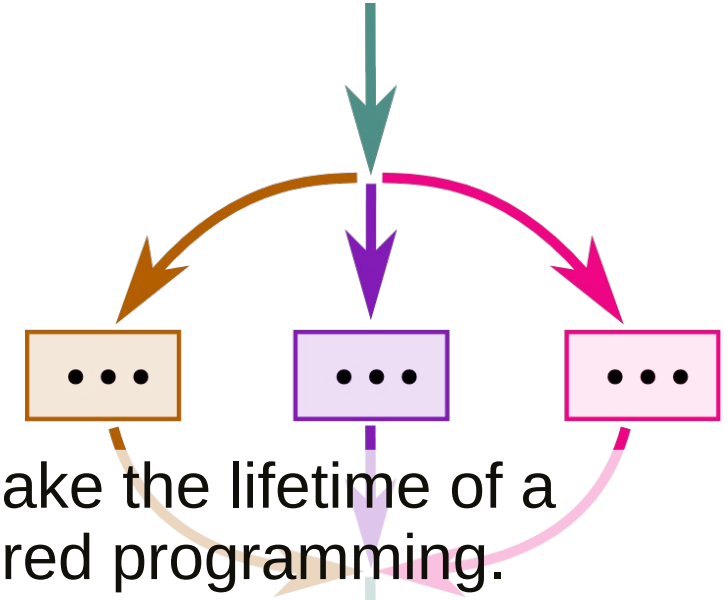
Structured Concurrency



Creating a system thread is expensive because the call takes up much time, and each thread takes up some memory. Your threads also share the same CPU, so you don't want them to block it, causing other threads to wait unnecessarily.

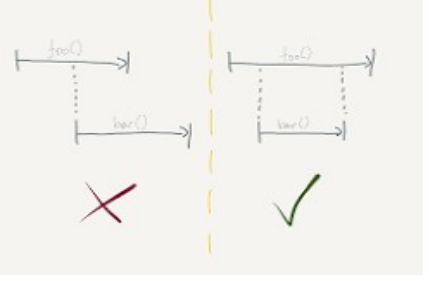
- Main task splits into several concurrent tasks
- Spawning threads must terminate before the main thread
- Executors API to create an ExecutorService that starts a new virtual thread for each task.
- ExecutorServices API Executors.newVirtualThreadExecutor()
- ExecutorService defines submit methods to execute tasks for execution. The submit methods don't block, instead they return a Future object that can be used to wait for the result or exception.
- No impact on ForkJoinPool APIs
- *No leaked threads, no leaked exceptions!*

Structured Concurrency



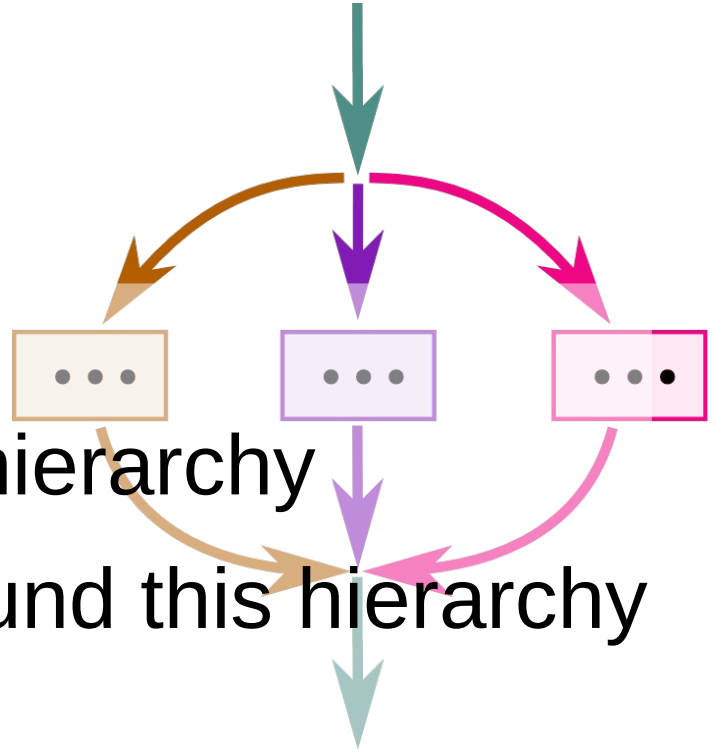
The idea behind structured concurrency is to make the lifetime of a thread work the same as code blocks in structured programming.

For example, in a structured programming language like Java, If you call method **B** inside method **A** then method **B** must be finished before you can exit method **A**. The lifetime of method **B** can't exceed that of method **A**.



Structured Concurrency

- Bind thread lifetime to a scope
- Interpret this as a parent – child hierarchy
- Build programming concepts around this hierarchy



Structured Concurrency I

```
var NTASKS = 1000;

try (ExecutorService exec = Executors.newVirtualThreadExecutor()) {
   for (int i = 0; i < NTASKS; i++) {
      exec.submit(() -> {
          try { TimeUnit.SECONDS.sleep(1);}
          catch (InterruptedException e) {}});
   }
}

// Blocks until all threads completed,
// ExecutorService implements AutoClosable
```

```
ExecutorService exec =
Executors.newVirtualThreadExecutor().withDeadline(
   Instant.now().plus(30, ChronoUnit.SECONDS));
```

```
CompletableFuture<Integer> future = exec.submitTask(callable);
```

Structured Concurrency II

```
Response handle() throws ExecutionException, InterruptedException {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        Supplier<String>  user  = scope.fork(() -> findUser());
        Supplier<Integer> order = scope.fork(() -> fetchOrder());

        scope.join()            // Join both subtasks
             .throwIfFailed();  // ... and propagate errors

        // Here, both subtasks have succeeded, so compose their results
        return new Response(user.get(), order.get());
    }
}
```

Structured Task Scope

The general workflow of code using StructuredTaskScope is as follows:

1) Create a scope. The thread that creates the scope is its owner.
2) Use the fork(Callable) method to fork subtasks in the scope (*as a virtual thread)*.
3) At any time, any of the subtasks, or the scope's owner, may call the scope's shutdown() method to cancel unfinished subtasks and prevent the forking of new subtasks.
4) The scope's owner joins the scope, i.e., all of its subtasks, as a unit. The owner can call the scope's join() method, to wait until all subtasks have either completed or been cancelled via shutdown(). Alternatively, it can call the scope's joinUntil(java.time.Instant) method, to wait up to a deadline.
5) After joining, handle any errors in the subtasks and process their results.
6) Close the scope, usually implicitly via try-with-resources. This shuts down the scope, if it is not already shut down, and waits for any straggling subtasks to complete.

Loom Preparation Steps

- Virtual Threads (JEP 425) delivers virtual threads to Java and JVM
  - Build on the following JEP
    - JEP 416 (Reimplement Core Reflection with Method Handles) in JDK 18 removed the VM-native reflection implementation. This allows virtual threads to park gracefully when methods are invoked reflectively.
    - JEP 353 (Reimplement the Legacy Socket API) in JDK 13, and JEP 373 (Reimplement the Legacy DatagramSocket API) in JDK 15, replaced the implementations of java.net.Socket, ServerSocket, and DatagramSocket with new implementations designed for use with virtual threads.
    - JEP 418 (Internet-Address Resolution SPI) in JDK 18 defined a service-provider interface for host name and address lookup. This will allow third-party libraries to implement alternative *java.net.InetAddress* resolvers that do not pin threads during host lookup.

Loom Limitations

- There are still important limitations. In particular,
  - *blocking on monitors (java.lang.Object locks)*
  - Blocking on local file I/O is not yet retrofitted
- it will block the carrier thread.

Practice

- Write a structured concurrency program
- Debug the code
- Study the API
- *Reflect about various concurrency and synchronization approaches*