Evaluating Java Structured Concurrency

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand Java’s structured concurrency model
- Recognize the classes used to program Java’s structure concurrency model
- Evaluate the design & performance results of various Java concurrency models
- Learn how StructuredTaskScope is implemented
- Know how to implement a custom StructuredTaskScope
- Be able to evaluate the pros & cons of Java structured concurrency

Pros of Java Structured Concurrency

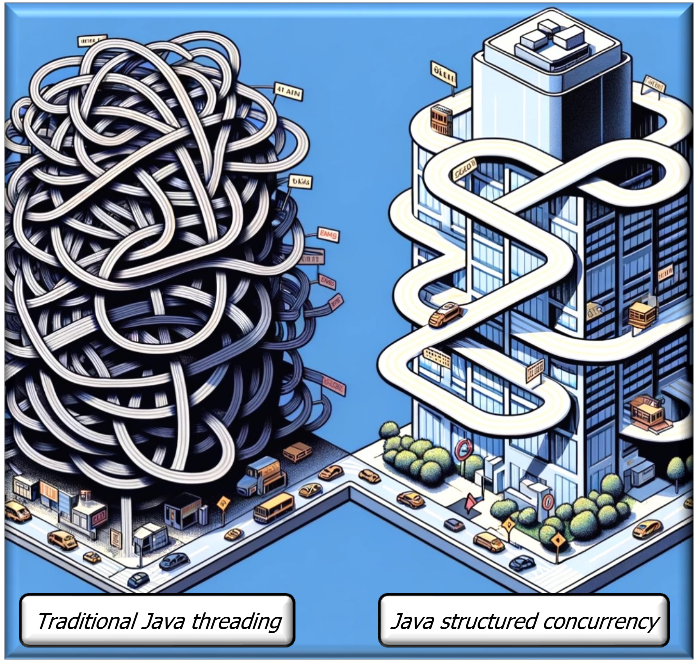
Pros of Java Structured Concurrency

- Benefits

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

    scope.join();
    scope.throwIfFailed();

    return new Response
      (user.resultNow(),
       order.resultNow());
}
```

Pros of Java Structured Concurrency

- Benefits
  - Provides greater clarity to the structure of concurrent code

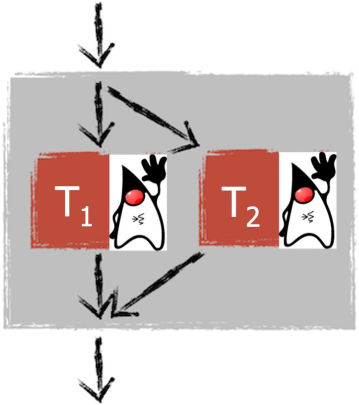


```
Response handle() ... {
  try (var scope = new
       StructuredTaskScope
       .ShutdownOnFailure()) {
    Future<String> user = scope
      .fork(() -> findUser());
    Future<Integer> order = scope
      .fork(() -> fetchOrder());

    scope.join();
    scope.throwIfFailed();

    return new Response
      (user.resultNow(),
       order.resultNow());
}
```

Particularly when compared with traditional Java "free threading" designs

Pros of Java Structured Concurrency

- Benefits
  - Provides greater clarity to the structure of concurrent code
    - Creates parent/child relationship between invoker & its subtasks



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

    scope.join();
    scope.throwIfFailed();

    return new Response
       (user.resultNow(),
        order.resultNow());
```

The handle() method is the parent task & the findUser() & fetchOrder() methods are its two children sub-tasks

Pros of Java Structured Concurrency

- Benefits
  - Provides greater clarity to the structure of concurrent code
    - Creates parent/child relationship between invoker & its subtasks

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

    scope.join();
    scope.throwIfFailed();

    return new Response
      (user.resultNow(),
       order.resultNow());
}
```

The whole block of handle() method code therefore becomes atomic

Pros of Java Structured Concurrency

- Benefits
  - Provides greater clarity to the structure of concurrent code
  - It enables short-circuiting in error handling

If one sub-task fails the other will be canceled if it's not completed yet

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

    scope.join();
    scope.throwIfFailed();

    return new Response
      (user.resultNow(),
       order.resultNow());
}
```

Pros of Java Structured Concurrency

- Benefits
  - Provides greater clarity to the structure of concurrent code
  - It enables short-circuiting in error handling
  - Supports interrupts

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

    scope.join();
    scope.throwIfFailed();

    return new Response
      (user.resultNow(),
       order.resultNow());
```

If parent task thread is interrupted before or during the join() call, both forks are canceled automatically at scope exit

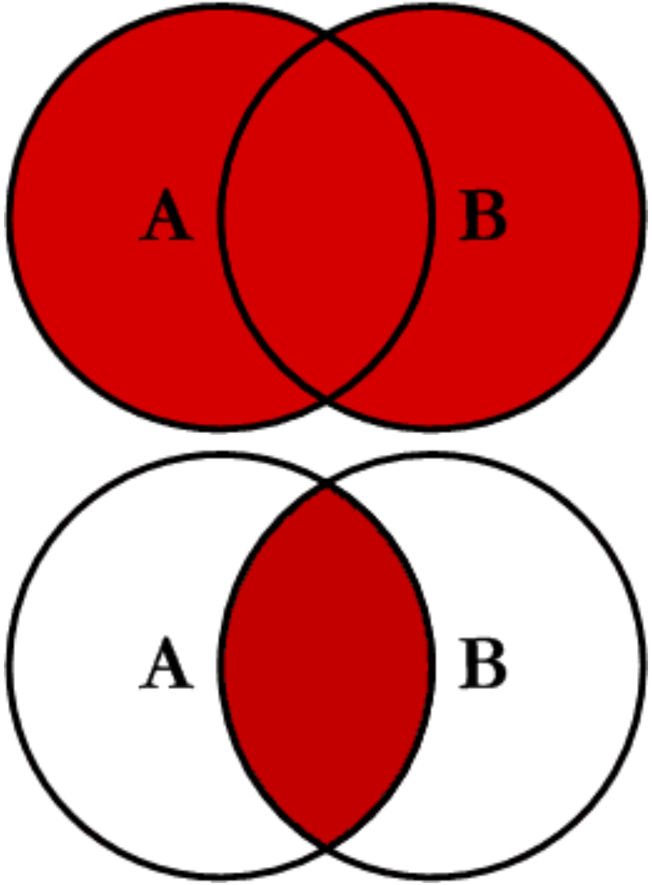
Pros of Java Structured Concurrency

- Benefits
  - Provides greater clarity to the structure of concurrent code
  - It enables short-circuiting in error handling
  - Supports interrupts
  - Easier to read & reason about the code
    - It looks like it's running in a single-threaded environment

```
Response handle() ... {
  String user = findUser();

  Integer order = fetchOrder();

  return new Response(user,
                      order);
}
```

Pros of Java Structured Concurrency

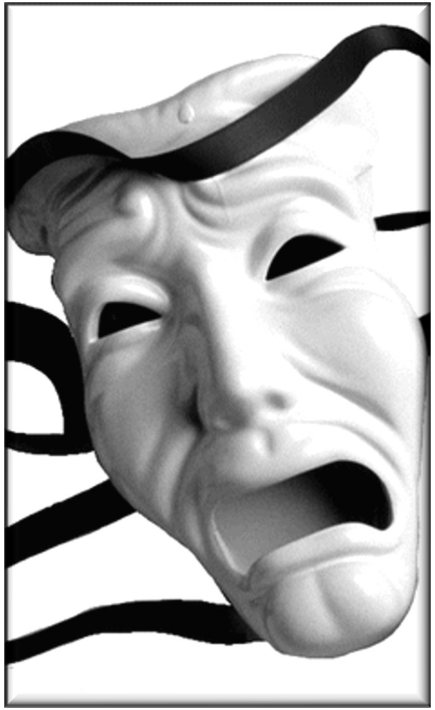
- Benefits
  - Provides greater clarity to the structure of concurrent code
  - It enables short-circuiting in error handling
  - Supports interrupts
  - Easier to read & reason about the code
  - Supports both "invoke-all" & "invoke-any" semantics



Cons of Java Structured Concurrency

Cons of Java Structured Concurrency

- Limitations



```
void sortAndPrintList
   (List<Future<BigFraction>> list) {
   try (var scope = ShutdownOnSuccess
     <List<BigFraction>>()) {
     Future<List<BigFraction>> qsF = scope
      .fork(() -> quicksort(list
             .stream()
             .map(Future::resultNow)
             .toList()));
     Future<List<BigFraction>> hsF = ...

     scope.join();

     printResult(scope.result());
   } catch (Exception ex) { ... }
}
```

Cons of Java Structured Concurrency

- Limitations
  - Use of Future is rather awkward & limiting

```
void sortAndPrintList
  (List<Future<BigFraction>> list) {
  try (var scope = ShutdownOnSuccess
    <List<BigFraction>>()) {
    Future<List<BigFraction>> qsF = scope
     .fork(() -> quicksort(list
           .stream()
           .map(Future::resultNow)
           .toList()));
    Future<List<BigFraction>> hsF = ...

    scope.join();

    printResult(scope.result());
  } catch (Exception ex) { ... }
}
```

Sort the list in parallel & print the results

Cons of Java Structured Concurrency

- Limitations
  - Use of Future is rather awkward & limiting

```
void sortAndPrintList
   (List<Future<BigFraction>> list) {
  try (var scope = ShutdownOnSuccess
    <List<BigFraction>>()) {
    Future<List<BigFraction>> qsF = scope
     .fork(() -> quicksort(list
                  .stream()
                  .map(Future::resultNow)
                  .toList()));
    Future<List<BigFraction>> hsF = ...

    scope.join();

    printResult(scope.result());
  } catch (Exception ex) { ... }
}
```

Need to convert List of Future objects to List of objects

Cons of Java Structured Concurrency

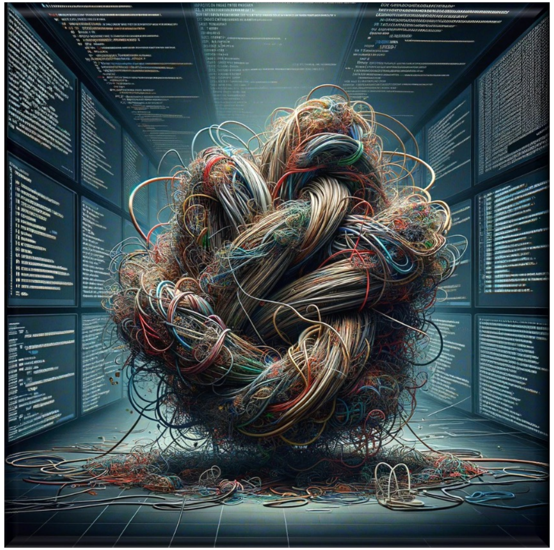
- Limitations
  - Use of Future is rather awkward & limiting

Cannot chain Future objects together (cf. Java CompletableFuture)

```
void sortAndPrintList
   (List<Future<BigFraction>> list) {
   try (var scope = ShutdownOnSuccess
     <List<BigFraction>>()) {
     Future<List<BigFraction>> qsF = scope
      .fork(() -> quicksort(list
             .stream()
             .map(Future::resultNow)
             .toList()));
     Future<List<BigFraction>> hsF = ...

     scope.join();

     printResult(scope.result());
   } catch (Exception ex) { ... }
}
```

Cons of Java Structured Concurrency

- Limitations
  - Use of Future is rather awkward & limiting
  - Syntax is rather verbose



```
void sortAndPrintList
   (List<Future<BigFraction>> list) {
  try (var scope = ShutdownOnSuccess
     <List<BigFraction>>()) {
     Future<List<BigFraction>> qsF = scope
       .fork(() -> quicksort(list
              .stream()
              .map(Future::resultNow)
              .toList()));
     Future<List<BigFraction>> hsF = ...

     scope.join();

     printResult(scope.result());
  } catch (Exception ex) { ... }
}
```

Cons of Java Structured Concurrency

- Limitations
  - Use of Future is rather awkward & limiting
  - Syntax is rather verbose
    - cf. Completable Future

```
void sortAndPrintList
  (List<<BigFraction> list) {
  var qsF = CompletableFuture
    .supplyAsync(() -> quicksort(list));

  var hsF = CompletableFuture
    .supplyAsync(() -> heapsort(list));

  qsF.acceptEither(hsF,
                   this::printResult)
     .handle(...)
     .join();
}
```

Sort the list in parallel & print the results

See www.baeldung.com/java-completablefuture

End of Evaluating Java Structured Concurrency