Enhanced design with lambdas

Raoul-Gabriel Urma
Richard Warburton

ITERATR LEARNING

Outline

1. Execute around: resource handling
2. Deferred execution: Logging
3. Using Lambdas in APIs

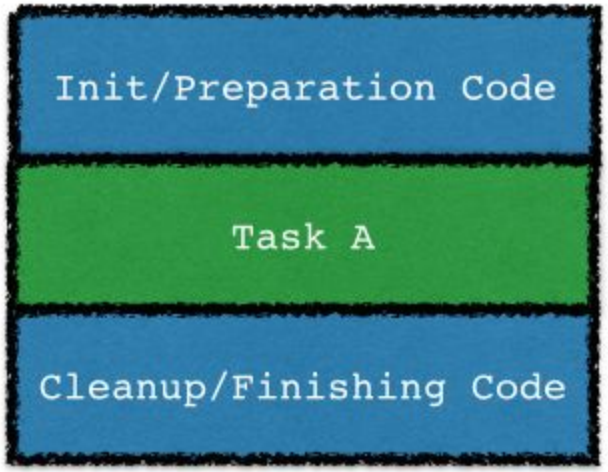
Execute around: resource handling

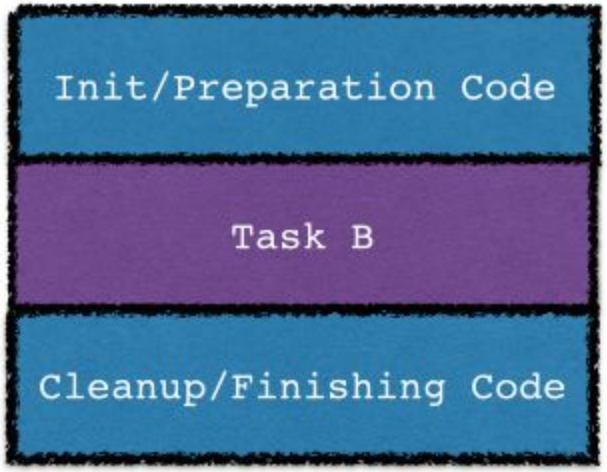
Example

```
public static String processFile() throws IOException {
 try (BufferedReader br =
   new BufferedReader(new FileReader("data.txt"))) {
     return br.readLine();
 } catch(IOException e) {
   logger.log(Level.SEVERE, "an exception was thrown", e);
   throw e;
 }
}
```

- What about returning two lines?
- What about returning the most frequent word?

Visually


Init/Preparation Code
Task A
Cleanup/Finishing Code


Init/Preparation Code
Task B
Cleanup/Finishing Code

Step 1: Recall Behaviour Parameterisation

- We want to achieve this:

```
String twoLines =
  processFile((BufferedReader br) ->
                br.readLine() + br.readLine());

String mostCommonWord =
  processFile((BufferedReader br) ->
                findMostCommonWord(br));
```

Step 2: Use a functional interface

```
@FunctionalInterface
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}

static String processFile(BufferedReaderProcessor p)
throws IOException {

}
```

Step 3: Execute a behavior!

```
String processFile(BufferedReaderProcessor processor)
throws IOException {
  try (BufferedReader br = new BufferedReader(new
    FileReader("data.txt"))) {
      return processor.process(br);
    }
  catch(IOException e) {
    logger.log(Level.SEVERE, "an exception was thrown", e);
    throw e;
  }
}
```

Processing the BufferedReader object

Step 4: Pass lambdas

Processing one line:

```
String oneLine =
    processFile((BufferedReader br) -> br.readLine());
```

Processing two lines:

```
String twoLines =
    processFile((BufferedReader br) -> br.readLine() +
       br.readLine());
```

Refactoring Practical

- get numbers from a file
- compute the binary string values of each number
- store these numbers in a cache
- See:
  - Pass:

    `com.java_8_training.problems.design.CacheInitializerTest`
  - Code: `com.java_8_training.problems.design.NumberCache`

Deferred execution: Logging

Logging

```
if (logger.isLoggable(Log.FINER)){
    logger.finer("Problem: " + generateDiagnostic());
}
```

- The state of the logger (what level it supports) is exposed in the client code through the method `isLoggable`.
- Why should you have to query the state of the logger object every time before you can log a message? It just clutters your code.

Logging: internal checking

```
logger.log(Level.FINER,"Problem: "+ generateDiagnostic());
```

- Code not cluttered with if checks anymore
- State of logger no longer exposed
- However logging message is always evaluated

Logging: using lambdas

```
public void log(Level level, Supplier<String> msgSupplier){
    if(logger.isLoggable(level)){
        log(level, msgSupplier.get());
    }
}

logger.log(Level.FINER, () -> "Problem: " + generateDiagnostic());
```

executing the lambda

defer construction of
logging message

Refactoring Practical

- Assertion framework with message diagnosis
- Inspired by Junit 5
- See:
  - Test: `com.java_8_training.problems.design.MiniAssertTest`
  - Code: `com.java_8_training.problems.design.MiniAssert`

Using Lambdas in APIs

The problem

- Lambdas can throw exceptions
- If your code uses a callback then it needs to take account of exception safety
- Is your object in the code state if the lambda throws an exception?

Guidelines

- Take care with callbacks
- Defer mutation until after execution of a callback using that data structure
- Consider wrapping callbacks in try/catch blocks.

Summary

Key Takeaways

- It’s not just about using Streams and Collectors
- Think about how to apply lambda expressions within your application
- They can influence the design and API approaches that you take.