Lesson 3-3: Avoiding The Use of forEach

Using Streams Effectively

Stop Thinking Imperatively

- Imperative programming uses loops for repetitive behaviour
- It also uses variables to hold state
- We can continue to do that in some ways with streams
- THIS IS WRONG

Stream Example

Still Thinking Imperatively

```
List<Transactions> transactions = ...

LongAdder transactionTotal = new LongAdder();

transactions.stream()
  .forEach(t -> transactionTotal.add(t.getValue()));

long total = transactionTotal.sum();
```

We are modifying state which is wrong for a functional approach

Stream Example

Now Using Correct Functional Approach

```
List<Transactions> transactions = ...

long total = transactions.stream()
   .mapToLong(t -> t.getValue())
   .sum();
```

Create a stream of long values that is passed to the next function

Use a reduction to create a single result

Legitimate Use of forEach

No State Being Modified

- Simplified iteration
- May be made parallel if order is not important

```
List<Transactions> transactions = ...

transactions.stream()
  .forEach(t -> t.printClientName());
```

Section 3

Summary

- If you are thinking of using `forEach()`, stop
- Can it be replaced with a combination of mapping and reduction?
- If so, it is unlikely to be the right approach to be functional
- Certain situations are valid for using `forEach()`
  - E.g. printing values from the stream

ORACLE®