Learning to Love the Lambda in the Stream

Introduction to Java Lambdas, Functional Interfaces, and Streams

Speaker Introduction

- These slides (web): https://tinyurl.com/lambda-web
- Richard Roda
- Sr. Developer at USANA Health Sciences
- Over 15 years of Java development experience
- Oracle Certified Professional Java 8
- Linked In: https://www.linkedin.com/in/richardroda
- These slides (pdf): https://tinyurl.com/love-lambda

What is a Lambda Expression?

- In Java, it is an unnamed function that is bound to a *functional interface* as an object.
- A *functional interface* is an interface with exactly one abstract method.
- Similar to an inner class: class members, *effectively final* arguments and local variables are available to it.
- An *effectively final* local variable or argument is either declared final, or is not changed such that if the final declaration were added, the code remains valid.
- Lambdas may only exist when assigned to a functional interface, including being passed in as a parameter or returned as a result.

Lambda Examples

- Example 1a

```
Predicate<Integer> isFive = n -> n == 5;
System.out.println(isFive.test(4)); // false
```

- Example 1b

```
// Higher order function that creates predicates.
Predicate<Integer> makeTestFunction(int value)
   { return n -> n == value; }
Predicate<Integer> isFour = mkTestFunc(4);
System.out.println(isFour.test(4)); // true
```

- Lamdba expressions must be assigned to a functional interface
- `(n -> n == 5).test(4); // Does not compile`
- `var unknownType = n -> n == 5; // Does not compile`
- `var predicateType = mkTestFunc(4);// Compiles`

Lambda Syntax

- *[Argument List]* **->** *[Statements]*
- Argument List may take one of the following forms:
  - () ->
  - i ->
  - (i) -> or Java 11+: (*@Annotations* **`var`** i) ->
  - (*@Annotations* Integer i) ->
  - (i,j…) -> or Java 11+: (*@Annotations* **`var`** i, *@Annotations* **`var`** j, …) ->
  - (*@Annotations* Integer i, *@Annotations* String j…) ->
- Statements may take one of the following forms:
  - -> *statement*
  - -> { *statement* … *statement*; **`return`** `result;` }
- *@Annotations* are zero or more parameter annotations.

Functional Interface (FI) in Java

- "A functional interface is any interface that contains only one abstract method." — Oracle Java Tutorial
- The sole abstract method is referred to as the *functional method*
- Example 2- Valid Functional Interface

```
@FunctionalInterface // Optional
public interface Example2 {
    int myMethod(); // Functional Method
    boolean equals(Object other); // Not abstract -- in Object
    int hashCode(); // Not abstract -- in Object
    default int myMethod2() {return myMethod();} // Has implementation
    static int myMethod3() {return 0;} // Static and has implementation
}
```

Inner Classes vs Lambda

```
Example2 innerClass = new Example2() {
            @Override
            public int myMethod() {
                return 2;
            }
        };
Example2 lambda = ()-> 2;
```

- Equivalent implementations of the Example2 interface.
- The lambda declaration has two key advantages:
  - It is a single, concise line of code.
  - Because it is self-contained, the compiler will automatically fold it into a single (static) implementation.

Key Functional Interfaces

Used by Streams

Functional Interface Conventions

- The abstract method is called the *functional method*
- The term "Functional Interface" may be abbreviated as "FI"
- The following conventions apply for type variables used by Java FIs:
  - T – First argument, R – Return Value, U – Second argument
  - Any of the above are omitted if not used or the same as T.
- Many FIs that take one argument have a corresponding two argument version prefixed with "Bi"
- Many generic FIs have related primitive FIs prefixed with Double, Int, and Long for the respective data types.

Predicate<T>

- Accepts an argument, returns a `boolean`.
- Commonly used to find a matching element, or filter for matching elements.
- Functional method: `boolean test(T t)`
- 2 argument FI: `BiPredicate<T,U>`
- Related Primitive FIs: `DoublePredicate`, `IntPredicate`, `LongPredicate`
- Collections have a removeIf method to remove all matching elements.

```
boolean removeIf(Predicate<? super E> filter)
```

Consumer<T>

- Accepts an argument. Returns no value (void).
- Commonly used to perform an operation, such as printing.
- Functional Method: `void accept(T t)`
- 2 Argument FI: `BiConsumer<T,U>`
- Related Primitive FIs: `DoubleConsumer`, `IntConsumer`, and `LongConsumer`
- Collections and Streams have a forEach method to apply an action to each of their elements:

```
void forEach(Consumer<? super T> action)
```

- Has side effects and never a pure function.

Supplier<T>

- Accepts no arguments, returns a result
- Commonly used to provide an initial value to an algorithm, and as a source for multiple values.
- Functional Method: `T get()`
- Related Primitive FIs: `DoubleSupplier`, `IntSupplier`, `LongSupplier`
- Does not require that a new object be created.
- Associated with object creation and constructors.
- Useful for implementing the Abstract Factory design pattern.

Function<T,R>

- Accepts an argument, returns a result.
- Commonly used to map one value to another value, or compute a result.
- Represents a mapping from its input to its output.
- Functional Method: `R apply(T t)`
- 2 Argument FI: `BiFunction<T,U,R>`
- Related Primitive FIs: `[Double,Int,Long]Function`, `[Double,Int,Long]To[Double,Int,Long]Function`, `To [Double,Int,Long]Function`, `To[Double,Int,Long]BiFunction`

UnaryOperator<T> & BinaryOperator<T>

- A specialization of function: Accepts an argument, returns the same type of result as its argument.
- Used to compute a result or map a value to the same type as the input.
- Functional Method: `T apply(T t)`
- 2 Argument FI: `BinaryOperator<T>`
- Related Primitive FIs: `[Double,Int,Long]UnaryOperator`, `[Double,Int,Long]BinaryOperator`
- `UnaryOperator<T>` extends `Function<T,T>`
- `BinaryOperator<T>` extends `BiFunction<T,T,T>`

Comparator<T>

- Accepts two arguments, and returns an integer.
- Used to compare objects, and to impose a *total ordering* on a collection of objects.
- Functional Method: `int compare(T lhs, T rhs)`
  - When lhs < rhs, returns < 0
  - When lhs = rhs, returns 0
  - When lhs > rhs, returns > 0
- Even though Comparator has been around since the early days, it is a functional interface that is used by the stream framework for sorting data.

Optional<T> - Alternative to Null

- Represents a value that may or may not exist.
  - of – Create an optional from a non-null value.
  - ofNullable –Create an optional from a possible null value. An empty optional is created from a null value.
  - isPresent – Returns true when a value is present.
  - ifPresent –Accepts a Consumer on a present value.
  - get – Return present value or throw NoSuchElementException
  - orElse – Return present value or a provided value.
  - orElseGet – Return present value or get a value from Supplier.
  - orElseThrow – Return present value or throw Exception from Supplier.
  - map – Apply a Function mapping on a present value.
  - filter – Tests a Predicate on a present value, returning an empty Optional when the test result is false.

Pure Commutative Functions

- Do not use any information outside of their argument(s).
- No side effects: nothing outside of the return value changes.
- For any given arguments *X* an equivalent value *Y* is always returned regardless of the argument ordering.
- For Functions: fn.apply(X).equals(fn.apply(X)) is always true.
- For Suppliers: s.get().equals(s.get()) is always true.
- For BiFunctions fn.apply(X, Y).equals(fn.apply(X, Y)) and fn.apply(Y, X).equals(fn.apply(X, Y)) are always true.
- Such functions are inherently safe.
- “Pure Function” usually means Pure Commutative Function.

Is It Pure Commutative? (Yes)

- `IntUnaryOperator addOne = x -> x + 1;`
  - A function with one or zero arguments is commutative.
- ```
  ToDoubleFunction<Employee> getSalary =
      employee -> employee.getSalary();
  ```
  - Property getters without side-effects are pure.
- `Supplier<Set<Integer>> getSet = () -> new HashSet<>();`
  - This Supplier is pure: it always creates an empty hash set.
- `IntBinaryOperator plus = (x, y) -> x + y;`
  - It is pure and commutative: 3 + 4 = 7 = 4 + 3.
  ```

Is it Pure Commutative? (No)

- `IntBinaryOperator minus = (x, y) -> x - y;`
    - It is pure but not commutative: 3 - 4 = -1 ≠ 1 = 4 - 3
- `IntPredicate testSet = x -> mySet.add(x);`
    - It has side effects and uses information outside of its arguments.
- `IntConsumer printConsumer = x -> System.out.println(x);`
    - Consumers are not pure functions because they have side effects.

Method Reference

Shorthand for lambdas that invoke a single method

Method Reference

- Shorthand for a Lambda that only calls a method
- Types of References
  - Static method, such as `String::valueOf`
  - Constructor reference, such as `StringBuilder::new`
  - Method on an instance, such as `System.out::println`
  - Instance method, such as `String::toUpperCase`
- Once familiar with syntax, these can often be read and understood faster.
- A method reference may always be transformed into a lambda, but a lambda may not always be transformed into a method reference.

Static Method Reference

- Example:

```
// public static valueOf(char[] data) method on String
Function<char[],String> valueOf = String::valueOf;
// Equivalent lambda expression
// valueOf = s -> String.valueOf(s);
String value = valueOf.apply(new char[] {'H','i'});
System.out.println(value); // Hi
```

- Arguments are bound in declaration order.

Constructor Reference

- Example:

```
// public StringBuilder() constructor on StringBuilder
Supplier<StringBuilder> supplier = StringBuilder::new;
// Equivalent lambda expression
// supplier = () -> new StringBuilder();
StringBuilder sb = supplier.get().append("Hi!");
System.out.println(sb); // Hi!
```

- Syntax similar to static method reference that creates a new object.
- Creates a new instance of the class, and returns it as the result.
- Must be bound to a functional interface with a compatible return type.
- Arguments are bound in declaration order.
- Supplier FI is canonically used for a constructor method reference.

Method Reference on an Instance

- Example:

```
// public void print(Object x) method on out's
PrintStream
Consumer<Object> printer = System.out::print;
// Equivalent lambda expression
// printer = i -> System.out.print(i);
printer.accept("We come in peace.");// We come in peace.
```

- class members, *effectively final* arguments and local variables may be used as a method reference on an instance.
- Arguments are bound in declaration order.

Instance Method Reference

- Example:

```
// public String toUpperCase() method on String
UnaryOperator<String> toUpper = String::toUpperCase;
// Equivalent lambda expression
// toUpper = s -> s.toUpperCase();
System.out.println(toUpper.apply("abc")); // ABC
```

- The first argument of the lambda becomes the instance the method reference operates on.
- The remaining arguments are bound in the order they occur.
- The first argument rule has significance when choosing the order of arguments for the “Bi” family of Functional Interfaces.

Streams

Not to be confused with IO Streams

What is a Java Stream?

- Abstraction for computation of elements.
- A computation structure, not a data structure.
- A stream consists of
  1. A data source
  2. Zero or more intermediate operations.
  3. A terminal operation, which starts the processing.
  4. (Optional) A close operation, to release any resources such a files.

A Data Source

- Can be anything that supplies data
  - A Collection
  - A file
  - An Iterated Function
  - Can Be Infinite
- Is Lazy
  - Only used when a *terminal operation* is applied to the stream.

Intermediate Operations

- Returns a stream with the operation appended.
- Are Lazy
  - Only used when a *terminal operation* is applied to the stream.
- Typical Intermediate operations
  - Filtering or finding items that match a predicate
  - Mapping items using a function
  - Skipping and limiting items processed. Can turn an infinite stream into a finite stream.
  - Reordering the items

A Terminal Operation

- Often returns a result such as a value or collection
  - A *reduction* produces a result from every stream element
- Is Eager
  - Starts the processing of elements from the data source through any Intermediate operations
  - A stream is a passive description of a data source and intermediate operations until a terminal operation is applied.
- Executes the stream
  - Any further operations except close() result in an IllegalStateException
  - Does **not** close the stream.
  - Use a try-with-resources block with Closable data source streams.

Streams are Like Factory Conveyor Belts

- The data source is the raw material to be processed.
- Adding the intermediate operations is like getting the workers into place. The terminal operation is like the worker who packages the finished product.
- Like a conveyor belt takes the result of the previous worker's changes to the next worker, a Stream takes the data source output or previous intermediate operation result as the input to the next intermediate or terminal operation.
- A conveyor belt doesn't start until all the workers are in place and ready. Likewise a stream doesn't start until all the intermediate operations and the terminal operation have been defined.
- Defining the terminal operation starts the processing. Once it is running, it can't be changed.

Breaking Down the Stream

```
int addPositive(Collection<Integer> numbers) {
    return numbers.stream() // Data Source
        .filter(i->i > 0) // Intermediate Operation
        .reduce(0, (i,sum) -> i+sum); // Terminal Operation
}
```

- All streams have a data source, zero or more intermediate operations, and a terminal operation.
- `numbers` collection is the data source.
- `filter` is an intermediate operation.
- `reduce` is the *reduction* terminal operation on the stream.
- A reduction processes all of the values in a given stream to a single value.
- Integer reduction examples: sum, average, median, min, and max.

Primitive Streams

- IntStream, LongStream, and DoubleStream
- They offer a performance benefit over the generic stream by avoiding boxing of primitive computations.
- They offer additional terminal operations, such as sum(), min(), max(), average(), and summaryStatistics().
- Can replace a traditional for loop with range and forEach.

```
IntStream.range(0, 10).forEach(System.out::println); // Print 0-9
```

- Use mapToInt, mapToLong, mapToDouble, and mapToObj to convert an existing stream to an IntStream, LongStream, DoubleStream, and Stream<T> respectively.
- Use the boxed() method to convert a primitive stream to its equivalent object stream by boxing the primitive values as follows:
  - IntStream to Stream<Integer>
  - LongStream to Stream<Long>
  - DoubleStream to Stream<Double>

Parallelism and Ordering

- Parallel streams may process multiple elements at a time.
- Sequential streams process a single element at a time.
- Ordered streams have a defined order.
- Unordered streams lack a defined order.
- Both sequential and parallel streams may be ordered, but only an ordered sequential stream guarantees actual encounter order.
  - Certain operations are only well defined for ordered streams, and impose additional overhead on ordered parallel streams.
- Pure commutative functions and operations work correctly with any parallelism and ordering.

Data Source Examples

- Collection
  - Collection.stream() creates a sequential stream
  - Collection.parallelStream() creates a parallel stream
  - Stream ordering determined by underlying collection ordering
    - List, Queue, SortedSet, and LinkedHashSet are ordered
    - HashSet is unordered
- Stream.of() - Array
  - Stream.of(T… values) creates a sequential ordered stream.
- File
  - Files.lines(Path path) creates a stream of Strings from a file.
  - File streams should be closed and used with try-with-resources.
- Iterated Function
  - Stream.iterate(T seed, UnaryOperator<T> function) creates a sequential ordered infinite stream starting with the provided seed and repeatedly applying the provided function to it.

Intermediate Operations

These Create a New Stream with the Operation Appended to It

Map

- Not to be confused with `java.util.Map`.
- Uses a `Function<T,R>` to apply a computation or mapping on stream elements.
- A pure function should be used if possible.
- May change the type of a stream by returning values of a different type.

```
public double totalSalary(Collection<Employee> employees) {
    return employees.stream()
        .map(Employee::getSalary).reduce(0.0, (i,sum) -> i+sum);
}
```

- Use flatMap to process functions that return Streams.

```
int sumListOfLists(List<List<Integer>> listOfLists) {
    // flatMap replaces an element with the contents of a stream.
    // mapToInt creates an IntStream from a Stream.
    return listOfLists.stream()
        .flatMap(List::stream).mapToInt(i->i).sum();
}
```

Filter

- The `filter` intermediate operation retains the contents of the stream where the `Predicate` is `true`.

```
double totalCommissionPayable(Collection<Associate> associates) {
    return associates.stream()
        .filter(Associate::isCommissionQualified)
        .mapToDouble(Associate::getCommissionEarned).sum();
}
```

- In this example, the total commission payable is computed by filtering for records that are qualified for commission payment and then summing the commission earned.

Limit and Skip – Infinite to Finite Stream

- Limit intermediate operation limits the values produced by a stream. An infinite stream becomes a finite stream.
- Skip intermediate operation skips the specified elements
- Not pure commutative. Undefined on unordered stream.
- Introduces overhead on a parallel stream.
- Order of these operations matters
  - Skip before limit – Skipped items not counted against limit
  - Skip after limit – Skipped items counted against limit
- ```
  IntStream.iterate(0, i -> i+1).skip(4).limit(6)
  .forEach(System.out::print); // 456789
  ```
- ```
  IntStream.iterate(0, i -> i+1).limit(6).skip(4)
  .forEach(System.out::print); // 45
  ```

Limit Unbounded Streams

- An unbounded stream is a stream that has no known upper limit on its elements. An infinite stream is a kind of unbounded stream.
- Unless an unbounded stream is intentionally infinite, it should always be limited to prevent hanging.
- Even if the stream “should” terminate it is still a good defensive programming practice to include a limit.
- A limit larger than the upper bound of what should be processed but small enough to stop processing in a reasonable amount of time should be used.
- A good starting point for a limit value is an order or two of magnitude (ten to a hundred times) more than the longest observed (or known possible) size.

Dangerous Unbounded Processing

- Dangerous
-
```java
public Optional<Widget> findBlueWidget() {
 // May never get to the end nor find a blue widget
 // Will not close any underlying stream resources.
    return getUnboundedStream().filter(widget ->
      "blue".equals(widget.getColor())).findAny();
}
```
- The stream has no upper limit on what is will process.
- The stream does not close any resources such as files
  - Note: terminal operations do *not* close a stream.

Safe Unbounded Processing

- Safe

```
public Optional<Widget> findBlueWidget() {
  // Will exit with Optional.empty() after 10000 widgets.
  try (Stream<Widget> unbounded = getUnboundedStream()) {
    return unbounded.limit(10000).filter(widget ->
        "blue".equals(widget.getColor())).findAny();
  }
}
```

- The limit intermediate operation ensures an exit from the stream.
- The try-with-resources ensures that any underlying resources are closed.
- When building a stream from a closable resource, use the `.onClose()` intermediate operation to register a handler that is called when the stream is closed to close any underlying resources.

Distinct

- Filters out any duplicate items according to the Object.equals method.
- Distinct objects should have a hashCode method that is *consistent with equals*. When a.equals(b) then a.hashCode() == b.hashCode().
- For ordered streams, the first of a given value is preserved.
- For streams known to be distinct, such as an unmapped stream from a set, this method passes the values through. Examples:
  - `getCollectionStream().distinct().map(i->i+1) // Better`
  - `getCollectionStream().map(i->i+1).distinct() // Worse`
  - The first example bypasses distinct processing when the collection is a set.
  - Once a known distinct stream is mapped, it is no longer known to be distinct.
- Avoid use with parallel streams.

```
IntStream.of(1,4,2,1,2,5,4,3).distinct()
    .forEach(i->System.out.print(" " + i));
/* 1 4 2 5 3 */
```

Sorted

- Sorts stream items. Resulting stream is an ordered stream.
- Supports parallel streams. Stable for sequential ordered streams.
  - Stable sort means ties (compare = 0) retain underlying stream ordering.
- Sorts using the *natural order* only when elements are Comparable

```
Stream.of(8,4,6,3,7,8,2,3,4).sorted()
    .forEach(i->System.out.print(" " + i));
/* 2 3 3 4 4 6 7 8 8 */
```

- Sorts using a comparator

```
Stream.of(8,4,6,3,7,8,2,3,4)
  .sorted((lhs,rhs)->rhs-lhs)
  .forEach(i->System.out.print(" " + i));
 /* 8 8 7 6 4 4 3 3 2 */
```

Unordered

- Removes the ordered constraint from an ordered stream.
- Improves the performance of a parallel ordered stream.
- Use on a parallel stream that does not rely on ordering.
- Pure commutative functions and operations always work.

```
int total = IntStream.of(4,3,6,7,8,5,6,7,8)
    .parallel().unordered().sum();
System.out.print(total); // 54
```

- No benefit to using unordered with a sequential stream.

Sequential and Parallel

- The sequential() intermediate operation makes a stream sequential.
- The parallel() intermediate operation makes a stream parallel.
- May be used to maximize performance by parallelizing a stream when it is most beneficial to do so.

```
try (Stream<Widget> unbounded = getUnboundedStream()) {
    return unbounded.limit(10000).parallel()
        .unordered().filter(widget ->
            "blue".equals(widget.getColor())).findAny();
}
```

- Making the stream parallel after the limit operation avoids the additional overhead of the parallel ordered limit operation.

takeWhile (Java 9+)

- Transforms the stream to include the elements that match the predicate stopping when an element does not match.

```
IntStream.of(0,1,2,3,4,2,1).takeWhile(i->i/4 == 0)
.forEach(i->System.out.print(" " + i));
/* 0 1 2 3 */
```

- Unlike filter, processing stops at number 4.
- Stream is empty if first element does not match.
- Not pure commutative. Undefined on unordered stream.
- Introduces overhead on a parallel stream.

dropWhile (Java 9+)

- Transforms the stream to skip the elements that match the predicate stopping when an element matches.

```
IntStream.of(0,1,2,3,4,2,1).dropWhile(i->i/4 == 0)
.forEach(i->System.out.print(" " + i));
/* 4 2 1 */
```

- Unlike filter, matching and skipping stops at number 4.
- Stream has all elements if first element does not match.
- Not pure commutative. Undefined on unordered stream.
- Introduces overhead on a parallel stream.

Intermediate Operations May Be Added Conditionally

- Consider this code:
-
```
public int addModulo(int[] data, Integer modulo) {
return IntStream.of(data)
 .filter(datum->modulo == null || (datum % modulo) == 0)
 .sum();
}
```
- When a null modulo is passed in, all elements will be processed
- Is there a way we can take advantage of the fact that all are processed when modulo is null?

Optimize By Filtering Conditionally

- The example on the previous slide may be optimized by conditionally adding the filter and unboxing modulo.
-
```
public int addModulo(int[] data, Integer modulo) {
 IntStream sumStream = IntStream.of(data);
 if (modulo != null) {
     final int mod = modulo; // Factor out unboxing of int.
     // Must re-assign because .filter returns a stream.
     sumStream = sumStream.filter(datum->(datum % mod) == 0);
 }
 return sumStream.sum();
}
```
- The check for null and unboxing of modulo is done only once. The resulting stream operation will be more performant.

Terminal Operations

Let's Get This Party Started. Let’s Get This Stream Processing

Terminal Operations

- count - A *reduction* that returns the number of elements in the stream. Never use on an infinite stream.
- reduce - Perform a *reduction* of the stream using a `BinaryOperator` to accumulate the elements. Never use on an infinite stream.
- anyMatch - Returns **`true`** and stops processing if any element matches the supplied `Predicate`, **`false`** otherwise. Empty Stream is **`false`**.
- allMatch - Returns **`false`** and stops processing if any element does **not** match the supplied `Predicate`, **`true`** otherwise. Empty Stream is **`true`**
- noneMatch - Returns **`false`** and stops processing if any element matches the supplied `Predicate` , **`true`** otherwise. Empty Stream is **`true`**.
- forEach - A `void` operation that presents each element to a `Consumer` for processing. Avoid use on an infinite stream.
- A reduction is an operation that computes a single value by processing all the values on the stream. Never reduce an infinite stream.

Reduction – Add a Collection of Numbers

- Given `Collection<Integer> numbers` that has integers from 1 to 1000, add the collection.
- Stream reduction (using a `BinaryOperator<Integer>`)

```
 return numbers.stream().reduce(0, (i,sum) -> i+sum);
// 500500
```

- The first argument to reduce is the identity value. For addition and counting, it is 0. For a multiplication it is 1, for strings it is “” (empty string). In this case X + 0 = X.
- The second argument to reduce is the reduction function. In this case the reduction adds the numbers together.
- The identity value is returned for empty streams or used as the second argument when the first stream value is processed.
- This reduction function is both pure and commutative.

Terminal Operations that return Optional<T>

- These terminal operations return an `Optional<T>` because no value exists in an empty stream.
- findFirst - produces the first element in a stream. Imposes additional overhead on an ordered parallel stream. Equivalent to findAny when used on an unordered stream.
- findAny - produces any element on the stream. It does not impose any overhead on parallel stream, but may produce differing values from the same stream.
- Min – produces the minimum element.
- Max – produces the maximum element.
- Operations findFirst and findAny are not pure commutative.

Terminal Operations May Be Invoked Conditionally

- Consider the add modulo example from earlier.
-
```java
public int addModulo(int[] data, Integer modulo) {
IntStream sumStream = IntStream.of(data);
if (modulo != null) {
    final int mod = modulo;
    sumStream = sumStream.filter(datum->(datum % mod) == 0);
}
return sumStream.sum();
}
```
- How can this function be changed to support an operation argument that can be “count” if the numbers should be counted, or “sum” if the numbers should be summed?

Terminal Operations May Be Invoked Conditionally

- The terminal operation may called conditionally after the stream has been built with its intermediate conditions.
- ```
  public int addOrCountModulo(int[] data, Integer modulo,
      String operation) {
  IntStream opStream = IntStream.of(data);
  if (modulo != null) {
       final int mod = modulo;
       opStream = opStream.filter(datum->(datum % mod) == 0);
  }
  // Use count or sum depending on the requested operation.
  return "count".equals(operation) ?
  (int)opStream.count() : opStream.sum();
  }
  ```
- These techniques provide a more elegant solution for providing multi-purpose processing than an “if-else” statement chain or “case” statements.
- The Strategy pattern can also be used to build and configure a stream.
  ```

Collector (Terminal Operation)

A Mutable Reduction That Creates an Object to Process All Stream Elements

Never Use on an Infinite Stream

Collections Collectors

- These collectors take the elements and add them to a collection.
- There are toList(), toSet(), and toCollection() collectors.
- In Java 16+ toList() is also a terminal operation for convenience.
- ```
  List<Integer> ints = IntStream.of(5,4,3,3,2,1,1).boxed()
  .collect(Collectors.toList());
  System.out.println(ints); /* [5, 4, 3, 3, 2, 1, 1] */
  ```
- ```
  Set<Integer> intSet = IntStream.of(5,4,3,3,2,1,1).boxed()
  .collect(Collectors.toSet());
  System.out.println(intSet); /* [1, 2, 3, 4, 5] */
  ```
- ```
  // Custom collection type with a sort applied to it.
  LinkedHashSet<Integer> sortedSet= IntStream.of(5,4,3,3,2,1,1)
  .boxed().sorted(Comparator.reverseOrder())
  .collect(Collectors.toCollection(LinkedHashSet::new));
  System.out.println(sortedSet); /* [5, 4, 3, 2, 1] */
  ```
  ```

Partition Collector

- The Partition collector uses a `Predicate<T>` to create a map with the keys **`false`** and **`true`**.
- Both the **`false`** and **`true`** key and value always exist in the map even if the corresponding value is not present. In such a case, the value is typically an empty collection, an empty optional, or a sum or count of 0.
- Use the predicate in the previous example to create a map with elements divisible by 4 and not divisible by 4.

```
Map<Boolean,Integer> summap = IntStream.range(0,1000).boxed()
.collect(Collectors.partitioningBy(i -> i%4==0,
Collectors.summingInt(i -> i)));
System.out.println(summap); // {false=375000, true=124500}
```

The `summingInt` collector is a *downstream collector*. It processes each classification (key) for the map In this case, it accepts the values of the partitioning by collector and produces a sum reduction of the values.

Grouping By Collector

- For the next example, consider the following stream producing function
- `static Stream<String> aboutJack() {`

```
return Stream.of("All","work","and","no","play","makes",
"jack","a","dull","boy","but","all","play","and","no",
"work","makes","jack","a","fool"); }
```

- Group each word by starting letter, in alphabetical order

```
aboutJack().sorted().collect(
Collectors.groupingBy(s -> s.charAt(0), TreeMap::new,
Collectors.toCollection(TreeSet::new)));
/* A=[All], a=[a, all, and], b=[boy, but], d=[dull],
f=[fool], j=[jack], m=[makes], n=[no], p=[play], w=[work] */
```

- The `Collectors.toCollection` is a *downstream collector.* It processes the elements for each classification (key) in the map.

Grouping By Concurrent

- Streams may be processed in parallel by using the `parallel` method using concurrent collectors and data structures.

```
aboutJack().parallel().collect( Collectors.groupingByConcurrent(
s -> s.charAt(0), ConcurrentSkipListMap::new,
Collectors.toCollection(ConcurrentSkipListSet::new)));
/* A=[All], a=[a, all, and], b=[boy, but], d=[dull], f=[fool],
j=[jack], m=[makes], n=[no], p=[play], w=[work] */
```

- Count the words

```
aboutJack().parallel().collect(Collectors.groupingByConcurrent(
s -> s, ConcurrentSkipListMap::new, Collectors.counting()));
/* All=1, a=2, all=1, and=2, boy=1, but=1, dull=1, fool=1, jack=2,
makes=2, no=2, play=2, work=2 */
```

Joining Collector

- A process where a stream of `CharSequence` is concatenated together to form a string.

```
static Stream<String> aboutJack() { return Stream.of(
"All","work","and","no","play","makes","jack","a","dull",
"boy","but","all","play","and","no","work","makes",
"jack","a","fool"); }
```

- Join this into words separated with a space:

```
aboutJack().collect(Collectors.joining(" "));
/* All work and no play makes jack a dull boy but all play
and no work makes jack a fool */
```

Teeing Collector (Java 12+)

- A downstream collector that processes every element through two collectors and then uses a merge BiFunction to produce a result.

```
Double[] salaries = {95_000d, 125_000d, 35_000d, 40_000d};
Range salaryRange =
    Stream.of(salaries).collect(Collectors.teeing(
      Collectors.minBy(Double::compare),
      Collectors.maxBy(Double::compare),
      (min, max)-> new Range(min.orElse(0d), max.orElse(0d))));
System.out.println("Salary range is " + salaryRange.getMin() +
" - " + salaryRange.getMax());
/* Salary range is 35000.0 - 125000.0 */
```

Execute Around and Loan Patterns

Separate the concerns of manipulating resources from program logic

Source: *Functional Programming in Java* book by Venkat Subramaniam

The Execute Around Pattern

- Pattern to eliminate boilerplate code by performing operations before and after an operation.
- Consider this code for using a lock:

```
Lock lock = getLock();
try {
    return doSomething();
} finally {
    lock.unlock();
}
```

- We would have better separation of concerns if we could separate the lock manipulation from the operation.

Apply the Execute Around Pattern

- The boilerplate lock and unlock code may be refactored like this:
-
```
<T> T useLock(Supplier<T> operation) {
      Lock lock = getLock();
      lock.lock();
      try {
          return operation.get();
      } finally {
          lock.unlock();
      }
  }
```
- Then using the lock can be accomplished in a single line of code:

```
useLock(()-> doSomething());
```

Loan Pattern

- Specialized version of the execute around pattern
  1. Obtains or allocates a resource
  2. Initializes it
  3. Invokes a user specified operation with the resource
  4. Cleans it up
  5. Returns or deallocates a resource

Apply the Loan Pattern

- This example applies the loan pattern for JDBC connections
- ```
  @FunctionalInterface interface SqlFunction<T,R> {
      R apply(T t) throws SQLException;
  }
  ```

```
<T> T useDb(SqlFunction<? super Connection, ? extends T> operation) throws SQLException {
    try (Connection conn = getJdbcConnection()) {
        return operation.apply(conn);
    }
}
```

- The JDBC connection can be used with a single line of code:

```
useDb(conn -> doDbOperation(conn));
```
```

AutoClosable Lambdas

Use try-with-resources with any class, and catch the close exception

AutoClosable is a Functional Interface

-
```
public interface AutoCloseable {
  void close() throws Exception;
}
```
- This interface is a functional interface (FI) because it has exactly one abstract method.
- The Functional Method is: `void close()`.
- The missing `@FunctionalInterface` annotation is unnecessary.

Use try-with-resources with any class Example: Close a Context

- In Java 7, try-with-resources was added to the language.
- Unfortunately, not every class that could benefit from it implemented it.
- Using Lambdas, anything can leverage try-with-resources.
-

```
public void useContext(Context ctx) throws Exception {
  try(AutoCloseable it = ctx::close) {
      doSomethingWithContext(ctx);
  }
}
```

Issues with the AutoClosable Functional Interface (FI)

- The close method throws Exception.
- The declared Exception will either need to be caught or processed.
- This may result in the code being littered with unnecessary catch statements.

Fixing the AutoClosable FI

- If we wrote our own Closable interface:
- ```
  public interface NamingClosable extends AutoCloseable {
    @Override public void close() throws NamingException;
  }
  ```
- Then we can write
- ```
  public void useContext(Context ctx) throws NamingException
  {
    try(NamingClosable it = ctx::close) {
        doSomethingWithContext(ctx);
    }
  }
  ```

Parameterizing AutoClosable Exceptions

- Using generics, it is possible to parameterize the checked exceptions that a sub-interface of AutoClosable may throw.
- This example demonstrates how to parameterize a single checked exception.
-
```
public interface CloseIt1<E extends Exception>
extends AutoCloseable {
    default void close() throws E { closeIt(); }
    void closeIt() throws E;
}
```
- The default `close()` method is necessary because applying the generic to an abstract `close()` method results in a compiler error when used in a try-with-resources statement.

Using the Parameterized FI

- Using `CloseIt1` from the previous slide:
- ```java
  public void useContext(Context ctx) throws NamingException
  {
    try(CloseIt1<NamingException> it = ctx::close) {
        doSomethingWithContext(ctx);
    }
  }
  ```
- The close method of the Context is bound to the CloseIt1 resource. The try-with-resources feature of Java does the heavy lifting of the resource exception processing.
  ```

Decorator Pattern

- One of the core patterns introduced in the *Design Patterns, Elements of Reusable Object Oriented Software* by Gamma, Helm, Johnson, and Vlissides.
- Pattern allows behavior to be added to an object dynamically, by decorating it, or wrapping it with another object of the same abstract type (such as an interface).
- This pattern may be leveraged to add capabilities to AutoClosables, such as exception handling.
- Since AutoClosable is a Functional Interface, the decorator may be expressed as a lambda.
- https://en.wikipedia.org/wiki/Decorator_pattern

Decorating the Close Lambda

- Consider the following code
  - Assume `NotClosedException` is an unchecked exception with an accessible constructor that takes a Throwable.

```
public interface CloseIt0 extends AutoCloseable {
    public void close() throws NotClosedException;
    public static CloseIt0 wrapAllException(AutoCloseable
    autoCloseable) {
        // Decorating with a lambda that wraps all Exceptions
        return () -> { try { autoCloseable.close(); }
        catch (Exception ex) { throw new NotClosedException(ex);}
        };
    }
}
```

Catching the Decorated Close Exception

- This close lambda is decorated to wrap any exceptions that occur within a `NotClosedException`. If no exception occurs within the body, this wrapped exception will be caught and processed by the catch clause. Otherwise, it will be a suppressed exception.

```
public void useContext(Context ctx) throws NamingException {
    try(CloseIt0 it = CloseIt0.wrapAllException(ctx::close)) {
        readSomethingFromContext(ctx);
    } catch (NotClosedException ex) {
        logger.log(Level.WARNING, ex.getCause().getMessage()
        , ex.getCause());
    }
}
```

The CloseIt Project

- Provides generic functional interfaces extending `AutoCloseable` to use as the target of try-with-resources lambdas. Supports 0-5 checked exceptions.
- Makes it easy to use try-with-resources for anything that needs cleanup. May replace the try-finally construct.
- Provides these decorators for handling close exceptions
  - Ignore – Pretend the exception never happened. Discard it.
  - Consume – Do something, such as log the exception, then discard.
  - Rethrow – Do something, such a log the exception, then throw it.
  - Rethrow When – Do something, then conditionally throw it.
  - Hide – Hide a checked exception from the compiler and throw it.
  - Wrap – Wrap the exception within another exception of a different type. This is also a form of the Adapter design pattern. https://en.wikipedia.org/wiki/Adapter_pattern.

Questions

- Oracle's Lambda Quick Start Tutorial: http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html
- These slides (pdf): https://tinyurl.com/love-lambda
- CloseIt: https://github.com/RichardRoda/closeit - com.github.richardroda.util:closeit:1.7
- This Project: https://github.com/RichardRoda/2017-CodePaLOUsa-Lambda
- My Linked In: https://www.linkedin.com/in/richardroda
- These slides license: CC BY 3.0 US license terms