Java 8 Lambdas & Streams

Adib Saikali

@asaikali

The Plan

- Lambdas
- Break
- Streams

Some of the early slides will provide incomplete definitions which will be completed by the end of the presentation!

Questions!

- Questions are always welcome
- Ask questions when you have them!

Quick Survey

- Raise your hand if you have programmed using lambda expressions in the past
- Raise your hand if you have programmed with streams in the past

Java 8 Lambdas

- **Concept**
- Syntax
- Functional Interfaces
- Variable Capture
- Method References
- Default Methods

Lambda History

- The mathematical theory of lambda calculus was developed in the 1930s by Alonzo Church
- It found its way into programming languages in the 1960s
- Most major programming languages have support for lambda expressions including C++, C#, JavaScript, Python, Ruby … etc
- We finally have lambda expressions in the Java programming language

Lambda Definition

- Define anonymous functions
- Can be assigned to variables
- Can be passed to functions
- Can be returned from functions

What are lambdas good for

- Form the basis of the functional programming paradigm
- Make parallel programming easier
- Write more compact code
- Richer data structure collections
- Develop cleaner APIs

Concept vs. Implementation

- A concept can have multiple implementations
  - Map concept implemented by HashMap, ConcurrentHashMap
  - List concept implemented by ArrayList, LinkedList, SkipList
- Lambdas are a concept that has its own implementation flavor in different languages
- There are two distinct things to learn
  - The general concept of lambdas expressions
  - The Java 8 implementation of lambda expressions

Learning Tip

If this is your first encounter with Lambda expressions recognize that the learning curve associated with lambdas will be steeper for you than for some one who has already learned lambda expressions in another programming language.

Java 8 Lambdas

- Concept
- **Syntax**
- Functional Interfaces
- Variable Capture
- Method References
- Default Methods

Print a list of integers

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);

integers.forEach(
   x -> System.out.println(x)
);
```

Output:
1
2
3
4
5

forEach is a method that accepts a function as its input and calls the function for each value in the list

```
x -> System.out.println(x)
```

is lambda expression that defines an anonymous function with one parameter named x of type Integer.

Multi line Lambda

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);

integers.forEach(x -> {
        x = x + 10;
        System.out.println(x);
});
```

Output:

```
11
12
13
14
15
```

Lambda with a Local Variable

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);

integers.forEach((x) -> {
        int y = x / 2;
        System.out.println(y);
});
```

Output:

0
1
1
2
2

Specify Lambda Parameter Types

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);

integers.forEach((Integer x) -> {
        x = x + 10;
        System.out.println(x);
});
```

Output:

```
11
12
13
14
15
```

Lambda Expression Lifecycle

- Think of a lambda expression as having a two stage lifecycle.
  - Convert the lambda expression to a function
  - Call the generated function

```
x -> System.out.print(x);
```

```
public static void generatedNameOfLambdaFunction(Integer x){
  System.out.println(x);
}
```

What is the type of a lambda?

Type of a Lambda Expression

- Many programming languages define a function data type to represent lambda expressions
- Java 8 designers considered adding a function data type to Java 8 but rejected it because it was infeasible for a variety of reasons
  - Watch ***Lambda: A peek under the Hood*** by Brian Goetz for the details http://goo.gl/PEDYWi

Implementation Problems

```
x -> System.out.print(x);
```

```
public static void generatedNameOfLambdaFunction(Integer x){
  System.out.println(x);
}
```

- What class should the translated lambda expression function be placed in?
- How are instances of this class created?
- Should the generated method be a static or instance method?

Java 8 Lambdas

- Concept
- Syntax
- **Functional Interfaces**
- Variable Capture
- Method References
- Default Methods

Functional Interface

```
public interface Consumer<T> {
      void accept(T t);
}
```

- A functional interface is a regular Java interface with a single method
- This is a common idiom in existing Java code

JDK Functional Interfaces

```
public interface Runnable {
    public abstract void run();
}

public interface Comparable<T> {
    public int compareTo(T o);
}

public interface Callable<V> {
    V call() throws Exception;
}
```

Spring Functional Interfaces

```
public interface TransactionCallback<T> {
    T doInTransaction(TransactionStatus status);
}

public interface RowMapper<T> {
    T mapRow(ResultSet rs, int rowNum) throws SQLException;
}

public interface StatementCallback<T> {
    T doInStatement(Statement stmt) throws SQLException,
     DataAccessException;
}
```

Assign Lambda to a local variable

```
public interface Consumer<T> {
      void accept(T t);
}

void forEach(Consumer<Integer> action) {
  for (Integer i : items) {
     action.accept(t);
   }
}
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);

Consumer<Integer> consumer = x -> System.out.print(x);
integers.forEach(consumer);
```

- The type of a lambda expression is same as the functional interface that the lambda expression is assigned to!

Is Lambda an Anonymous Inner Class?

NO

We need a few more slides to develop the details

@FunctionalInterface

```
@FunctionalInterface
public interface PasswordEncoder {
      public String encode(String password, String salt);
}
```

- @FunctionalInterface causes the Java 8 complier to produce an error if the interface has more than one method

@FunctionalInterface

```
@FunctionalInterface
public interface PasswordEncoder {
      public String encode(String password, String salt);
      public String doSomethingElse();
}
```

- Produces a complier error
  Invalid *@FunctionalInterface* annotation; PasswordEncoder is not a functional interface
- @FunctionalInterface is an optional annotation.

Lambda & Backward Compatibility

- Any interface with one method is considered a functional interface by Java 8 even if was complied it with a Java 1.0 complier
- Java 8 lambdas will work with older libraries that use functional interfaces without any need to recompile or modify old libraries.

Backward Compatibility Example

```
JdbcTemplate template = getTemplate();
List<Product> products = template.query("SELECT * from products",
        new RowMapper<Product>(){
                @Override
                public Product mapRow(ResultSet rs, int rowNum) throws
      SQLException {

                        Integer id = rs.getInt("id");
                        String description = rs.getString("description");
                        Integer quantity = rs.getInt("quantity");
                        BigDecimal price = rs.getBigDecimal("price");
                        Date availability = rs.getDate("available_date");

                        Product product = new Product();
                        product.setId(id);
                        product.setDescription(description);
                        product.setQuantity(quantity);
                        product.setPrice(price);
                        product.setAvailability(availability);

                        return product;
                }
        });
```

Backward Compatibility Example

```
JdbcTemplate template = getTemplate();
List<Product> products = template.query("SELECT * from products",
        (ResultSet rs, int rowNum) -> {

                Integer id = rs.getInt("id");
                String description = rs.getString("description");
                Integer quantity = rs.getInt("quantity");
                BigDecimal price = rs.getBigDecimal("price");
                Date availability = rs.getDate("available_date");

                Product product = new Product();
                product.setId(id);
                product.setDescription(description);
                product.setQuantity(quantity);
                product.setPrice(price);
                product.setAvailability(availability);

                return product;
        });
```

Is this clear? Questions?

```
@FunctionalInterface
public interface PasswordEncoder {
      public String encode(String password, String salt);
}

public PasswordEncoder makeBadEncoder() {
            return (password, salt) -> password.toUpperCase();
}

public void doSomething(PasswordEncoder encoder) {
      String salted = encoder.encode("abc", "123");
}

PasswordEncoder encoder = makeBadEncoder();
doSomething(encoder);
```

Summary Please Read …

- A functional interface is an interface with a single method (incomplete definition)
- The method generated from a lambda 8 expression must have the same signature as the method in the functional interface
- In Java 8 the type of a Lambda expression is the same as the the functional interface that the lambda expression is assigned to.

Java 8 Lambdas

- Concept
- Syntax
- Functional Interfaces
- **Variable Capture**
- Method References
- Default Methods

Variable Capture

- Lambdas can interact with variables defined outside the body of the lambda
- Using variables outside the body of a lambda expression is called variable capture

Capture a local variable

```
public class LambdaCaptureExample {
  public static void main(String[] args) {
      List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);

      int var = 10;
      integers.forEach(x -> System.out.println(x + var));
      }
}
```

Output:

11
12
13
14
15

Effectively Final

```
public class LambdaCaptureExample {
  public static void main(String[] args) {
      List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);

          int var = 1;
          integers.forEach(x -> {
              var++;
Error: Local variable var defined in an enclosing scope must be final or effectively final
              System.out.println(x);
          });
      }
}
```

- Local variables used inside the body of a lambda must be declared final or the compiler must be able to tell that they are effectively final because their value is not modified elsewhere

Effectively Final

```
public class LambdaCaptureExample {
  public static void main(String[] args) {
      List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);

      int var = 1;
      integers.forEach(x -> System.out.println(x + var));
Error: Local variable var defined in an enclosing scope must be final or effectively final
      var = 50;
      }
}
```

Capture Static Variables

```
public class LambdaCaptureExample {
 private static int var = 1;

 public static void main(String[] args) {
      List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);
      integers.forEach(x -> System.out.println(x + var));
  }
}
```

Capture Class Variables

```
public class LambdaCaptureExample {
 private int var = 1;
 private List<Integer> integers = Arrays.asList(1,2,3,4,5);

 private void doSomething() {
  integers.forEach(x -> System.out.println(x + var));
 }

 public static void main(String[] args) {
  new LambdaCaptureExample5().doSomething();
 }
}
```

this within a lambda body

```
public class Example {
 private static final Example INSTANCE = new Example();
 private int var = 1;
 private List<Integer> integers = Arrays.asList(1,2,3,4,5);

 public void doSomething() {
        integers.forEach(x -> {
                System.out.println(x + this.var);
                if (this == INSTANCE) {
                        System.out.println("Within the lambda body this refers to the this of the
enclosing object");
                        }
                });
  }
  public static void main(String[] args) {
        INSTANCE.doSomething();
  }
}
```

This within a anonymous Inner class

```
public class Example {
 private static final Example INSTANCE = new Example();
 private int var = 1;
 private List<Integer> integers = Arrays.asList(1,2,3,4,5);

 public void doSomething() {
      integers.forEach( new Consumer<Integer>(){

                  private int state=10;

                  @Override
                  public void accept(Integer x) {
                        int y = this.state + Example.this.var + x;
                        System.out.println("Anonymous class " + y);
                  }

            });
  }
  public static void main(String[] args) {
      INSTANCE.doSomething();
  }
}
```

Lambda vs. Anonymous Inner Class

- Inner classes can have state in the form of class level instance variables lambdas can not
- Inner classes can have multiple methods lambdas only have a single method body
- **this** points to the object instance for an anonymous Inner class but points to the enclosing object for a lambda
- Lambda != Anonymous inner class

java.util.function.*

- **java.util.function** package contains 43 commonly used functional interfaces
  - **Consumer<T>** - function that takes an argument of type T and returns void
  - **Supplier<T>** - function that takes no argument and returns a result of Type T
  - **Predicate<T>** - function that takes an argument of type T and returns a boolean
  - **Function<T, R>** - function that takes an argument of Type T and returns a result of type R
  - … etc

Java 8 Lambdas

- Concept
- Syntax
- Functional Interfaces
- Variable Capture
- **Method References**
- Default Methods

Method References

- A lambda is a way to define an anonymous function
- But what if the function that we want to use is already written
- Method references can be used to pass an existing function in place where a lambda is expected

Reference a static method

```
public class Example {

    public static void doSomething(Integer i)
    {
        System.out.println(i);
    }

    public static void main(String[] args) {
        Consumer<Integer>  conusmer1 = x -> doSomething(x);
        conusmer1.accept(1); // 1

        Consumer<Integer> conusmer2 = Example::doSomething;
        conusmer2.accept(1); // 1

    }
}
```

The signature of the referenced method needs to match the signature of functional interface method

Reference a constructor

```
Function<String, Integer> mapper1 = x -> new Integer(x);
System.out.println(mapper1.apply("11")); // new Integer(11)

Function<String, Integer> mapper2 = Integer::new;
System.out.println(mapper2.apply("11")); // new Integer(11)
```

- Constructor methods references are quite handy when working with streams.

references to a specific object instance method

```
Consumer<Integer> conusmer1 = x -> System.out.println(x);
conusmer1.accept(1); // 1

Consumer<Integer> conusmer2 = System.out::println;
conusmer2.accept(1); // 1
```

- System.out::println method reference tells the compiler that the lambda body signature should match the method println and that the lambda expression should result in a call to System.out.println(x)

Instance method references to arbitrary object of a particular type

```
Function<String, String> mapper1 = x -> x.toUpperCase();
System.out.println(mapper1.apply("abc")); // ABC

Function<String, String> mapper2 = String::toUpperCase;
System.out.println(mapper2.apply("def")); // DEF
```

- Invoked on an object passed as input to the lambda

Method References Summary

Method Reference Type	Syntax	Example
Static	ClassName::StaticMethodName	String::valueOf
Constructor	ClassName::new	String::new
Specific object instance	objectReference::MethodName	System.out::println
Arbitrary object of a particular type	ClassName::InstanceMethodName	String::toUpperCase

Java 8 Lambdas

- Concept
- Syntax
- Functional Interfaces
- Variable Capture
- Method References
- **Default Methods**

The interface evolution problem!

- A natural place to use lambdas is with the Java collections framework
- The collection framework is defined with interfaces such as Iterable, Collection, Map, List, Set, … etc.
- Adding a new method such forEach to the Iterable interface will mean that all existing implementations of Iterable will break
- How can published interfaces be evolved without breaking existing implementations!

default Method

- A default method on java interface has an implementation provided in the interface and is inherited by classes that implement the interface.

```
public interface Iterable<T> {
 Iterator<T> iterator();

 default void forEach(Consumer<? super T> action) {
   Objects.requireNonNull(action);
   for (T t : this) {
     action.accept(t);
   }
 }

 default Spliterator<T> spliterator() {
   return Spliterators.spliteratorUnknownSize(iterator(), 0);
 }
}
```

default method inheritance

```
public interface Test {
	default void doSomething()
	{
		System.out.println("Test");
	}
}


public class TestImpl implements Test {

	public static void main(String[] args) {
		TestImpl1 testImpl = new TestImpl1();
		testImpl.doSomething(); // Test
	}
}
```

Overriding a default method

```
public interface Test {
	default void doSomething()
	{
		System.out.println("Test");
	}
}
public class TestImpl implements Test {

	@Override
	public void doSomething() {
		System.out.println("TestImpl");
	}

	public static void main(String[] args) {
		TestImpl testImpl = new TestImpl();
		testImpl.doSomething(); // TestImpl
	}
}
```

A hierarchy of default methods

```
public interface Test {
 default void doSomething(){
       System.out.println("Test");
 }
}

public interface TestA extends Test {
 @Override
 default void doSomething()
 {
       System.out.println("TestA");
 }
}

public class TestImpl implements TestA {
       public static void main(String[] args) {
             TestImpl testImpl = new TestImpl();
             testImpl.doSomething(); // TestA
       }
}
```

default method conflict

```
public interface A {
  default void doSomething() {
            System.out.println("A");
      }
}
```

```
public interface B {
  default void doSomething() {
            System.out.println("B");
      }
}
```

```
public class ABImpl implements A, B {

}
```

Compile Error: Duplicate default methods named doSomething with the parameters () and () are inherited from the types B and A

Resolving default method conflict

```
public interface A {
  default void doSomething() {
            System.out.println("A");
      }
}
```

```
public interface B {
  default void doSomething() {
            System.out.println("B");
      }
}
```

```
  public class ABImpl implements A, B {
        @Override
        public  void doSomething() {
              System.out.println("ABImpl");
              A.super.doSomething(); // notice A.super syntax
        }

        public static void main(String[] args) {
              new ABImpl2().doSomething(); // ABImpl \n A
        }
  }
```

Diamonds are No Problem

```
public interface A {
  default void doSomething() {
            System.out.println("A");
      }
}
public interface C extends A {
      default void other() {
            System.out.println("C");
      }
}
```

```
public interface D extends A {
 @Override
 default void doSomething(){
            System.out.println("D");
  }
}
```

```
  public class CDImpl implements C, D {
        public static void main(String[] args) {
              new CDImpl().doSomething();
        }
  }
```

Outputs : D

default method Summary

- A default method on an interface can have an implementation body
- if there is a conflict between default methods the class that is implementing the default methods must override the conflicting default method
- In an inheritance hierarchy with default methods the most specific default method wins.

Functional Interface Complete Definition

- A functional interface can only have one non default method

Are these concepts clear?

- Lambda expression
- Functional interface
- Method reference
- Default method

5 min break

- Start 5 minute Count Down Timer
- Streams

Collections Enhancements

- Java 8 uses lambda expressions and default methods to improve the existing java collections frameworks
- Internal iteration
  - delegates the looping to a library function such as forEach and the loop body processing to a lambda expression.
  - allows the library function we are delegating to implement the logic needed to execute the iteration on multiple cores if desired

Some New Methods

- New java.lang.Iterable methods in Java 8
  - default void forEach(Consumer<? super T> action)
  - default Spliterator<T> spliterator()
- New java.util.Collection Methods in Java 8
  - default boolean removeIf(Predicate<? super E> filter)
  - default Spliterator<E> spliterator()
  - default Stream<E> stream()
  - default Stream<E> parallelStream()

Some New Methods ….

- New java.util.Map Methods in Java 8
  - default V getOrDefault(Object key, V defaultValue)
  - putIfAbsent(K key, V value)
  - …. etc

Stream related methods

- Common stream related methods
  - default Spliterator<T> spliterator()
  - default Stream<E> stream()
  - default Stream<E> parallelStream()

What is a Stream?

- Streams are a functional programming design pattern for processing sequences of elements sequentially or in parallel
- When examining java programs we always run into code along the following lines.
  - Run a database query get a list of objects
  - Iterate over the list to compute a single result
  - Iterate over the list to generate a new data structure another list, map, set or … etc.
- Streams are a concept that can be implemented in many programming languages

Java 8 Streams

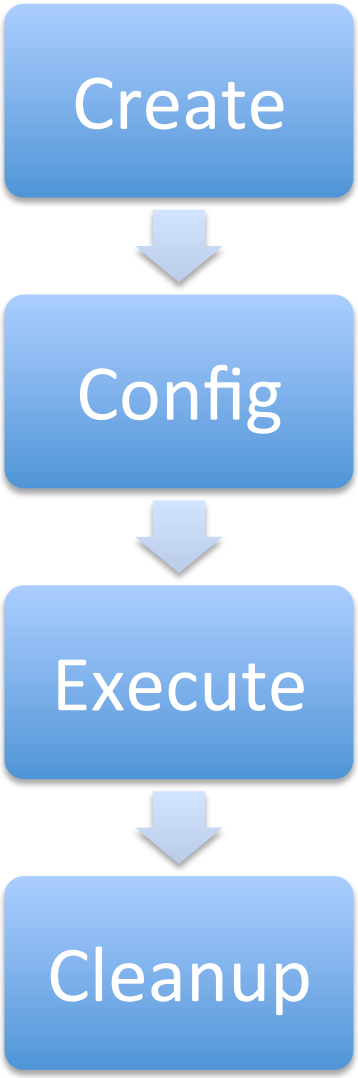
```
List<Order> orders = getOrders();
int totalQuantity  = orders.stream()
                .filter(o -> o.getType() == ONLINE)
                .mapToInt(o -> o.getQuantity())
                .sum();
```

- Create a stream instance from a **source** java collection
- Add a filter operation to the stream **intermediate operations pipeline**
- Add a map operation to to the stream **intermediate operations pipeline**
- Add a **terminal operation** that kicks off the stream processing

Stream Structure

- A stream has a
  - **source** that the stream can pull objects from
  - **pipeline** of operations that will execute on the elements of the stream
  - **terminal** operation that pulls values down the stream

Stream lifecycle



- **Creation** Streams get created from a source object such as a collection, file, or generator
- **Configuration** Streams get configured with a collection of pipeline operations.
- **Execution** Stream terminal operation is invoked which starts pulling objects trough the operations pipeline of the stream.
- **Cleanup** Stream can only be used once and have a close() method which needs to be called if the stream is backed by a IO source

Number Stream Source

```
System.out.println("Long Stream Source");
LongStream.range(0, 5)
                .forEach(System.out::println);
```

Output:

```
Long Stream Source
0
1
2
3
4
```

Collection Stream Source

```
List<String> cities = Arrays.asList("toronto",
      "ottawa","montreal","vancouver");
cities.stream()
     .forEach(System.out::println);
```

Output:

```
toronto
ottawa
montreal
vancouver
```

Character Stream Source

```
long length = "ABC".chars().count(); // <1>
System.out.println(length);
```

Output:

```
3
```

File system streams

```
String workingDir = System.getProperty("user.dir");
Path workingDirPath =
    FileSystems.getDefault().getPath(workingDir);

System.err.println("Directory Listing Stream\n");
Files.list(workingDirPath)
    .forEach(System.out::println);

System.err.println("Depth First Directory Walking Stream\n");
Files.walk(workingDirPath)
    .forEach(System.out::println);
```

What does this program do?

```
public class Example {
  public static void main(String[] args) throws IOException {

        String workingDir = System.getProperty("user.dir");
        Path workingDirPath = FileSystems.getDefault().getPath(workingDir);

        String classFileName = Example.class.getName().replace('.', '/') +
        ".java";

        int maxDepth = 10;

        Files.find(workingDirPath, maxDepth,
                    (filename, attributes) -> filename.endsWith(classFileName)
         ).forEach(path -> {
                                try {
                                Files.lines(path).forEach(System.out::println);
                             } catch (Exception e) {}
                           });
        }
}
```

This program prints itself out when run in Eclipse

```
public class Example {
  public static void main(String[] args) throws IOException {

        String workingDir = System.getProperty("user.dir");
        Path workingDirPath = FileSystems.getDefault().getPath(workingDir);

        String classFileName = Example.class.getName().replace('.', '/') +
        ".java";

        int maxDepth = 10;

        Files.find(workingDirPath, maxDepth,
                        (filename, attributes) -> filename.endsWith(classFileName)
         ).forEach(
             path -> {
                         try {
                         Files.lines(path).forEach(System.out::println);
                    } catch (Exception e) {}
                        }
                      );
        }
}
```

Stream Structure

Stream Terminal Operations

- **Reduction Terminal Operations** — return a single result
- **Mutable Reduction Terminal Operations** — return multiple results in a container data structure
- **Search Terminal Operations** — return a result as soon as a match is found
- **Generic Terminal Operation** — do any kind of processing you want on each stream element
- Nothing happens until the terminal operation is invoked!

Reduction Terminal Operations

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);
Long count = integers.stream().count();
System.out.println(count); // 5

Optional<Integer> result;
result = integers.stream().min( (x, y) -> x – y);
System.out.println(result.get()) // 1

result = integers.stream().
                              max(Comparator.comparingInt(x -> x));
System.out.println(result.get()); // 5

Integer result = integers.stream().reduce(0,(x,y) -> x + y)
System.out.println(result); // 15
```

Mutable Reduction Operations

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5, 5);

Set<Integer> s = integers.stream().collect(Collectors.toSet());
System.out.println(set); // [1, 2, 3, 4, 5]

Integer[] a = integers.stream().toArray(Integer[]::new);
Arrays.stream(a).forEach(System.out::println);
```

- Collectors class defines many useful collectors
  - List, Set, Map, groupingBy,patitioningBy, … etc

Search Terminal Operations

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5, 5);

Optional<Integer> result = integers.stream().findFirst();
System.out.println(result.get()); // 1

boolean result = integers.stream().anyMatch(x -> x == 5);
System.out.println(result); // true

boolean result = integers.stream().anyMatch(x -> x > 3);
System.out.println(result); // false

Optional<Integer> result = integers.stream().findAny();
System.out.println(result.get()); ; // unpredictable result 1
```

Generic Terminal Operation

- forEach

Streams Pipeline Rules

- Streams can process elements sequentially
- Streams can process elements in parallel
- Therefore stream operations are **not allowed** to modify the stream source

Intermediate Stream Operations

- **Stateless intermediate operations** — do not need to know the history of results from the previous steps in the pipeline or keep track of how many results it have produced or seen
  - filter, map, flatMap, peek
- **Stateful Intermediate Operations** — need to know the history of results produced by previous steps in the pipeline or needs to keep track of how many results it has produced or seen
  - distinct, limit, skip, sorted

Stateless Intermediate Operations

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5, 5);

integers.stream().filter(x -> x < 4)
                 .forEach(System.out::println); // 1 \n 2 \n 3

List<Integer> m = integers.stream()
                          .map(x -> x + 1)
                          .collect(Collectors.toList());
m.forEach(System.out::println); // 2 3 4 5 6 6

IntSummaryStatistics stats = integers.stream()
                                     .mapToInt(x -> x)
                           .summaryStatistics()
System.out.println(stats);
// IntSummaryStatistics{count=6, sum=20, min=1, average=3.333333, max=5}
```

Stateful Intermediate Operations

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5, 5);

integers.stream().distinct()
                        .forEach(System.out::println);
// 1 2 3 4 5

integers.stream().limit(3)
                          .forEach(System.out::println);
// 1 2 3

integers.stream().skip(3)
                        .forEach(System.out::println);
// 4 5 5

List<Integer> integers = Arrays.asList(7,1,2,3,4,5, 5);
integers.stream().sorted()                    .forEach(System.out::println);
// 1 2 3 4 5 5 7
```

Do this make Sense?

```
Stream<Order> orderStream = OrderService.stream();
OptionalDouble average = orderStream
    .filter(o -> o.getItems().size() > 2)
    .mapToDouble(Order::total).average();

System.out.println(average.getAsDouble());

OptionalDouble average = orderStream
        .filter(o -> o.getItems().size() > 2)
            .mapToDouble( o -> o.getItems().stream().mapToInt(
                    item -> item.getQuantity() *
                item.getProduct().getPrice()).sum()
                )
            .average();
```

Parallel Streams

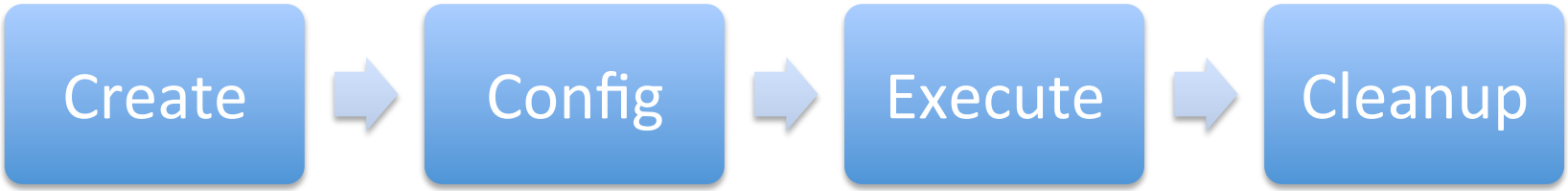
```
Stream<Order> orderStream = OrderService.parallelStream();
OptionalDouble average = orderStream
    .filter(o -> o.getItems().size() > 2)
    .mapToDouble(Order::total).average();

System.out.println(average.getAsDouble());
```

Summary

- Streams can be serial or parallel
- A stream structure

- Stream Lifecycle



Thank You