# Collections, Streams, and Filters

## Objectives

After completing this lesson, you should be able to:

- Describe the Builder pattern
- Iterate through a collection by using lambda syntax
- Describe the Stream interface
- Filter a collection by using lambda expressions
- Call an existing method by using a method reference
- Chain multiple methods
- Define pipelines in terms of lambdas and collections

## Collections, Streams, and Filters

- Iterate through collections using forEach
- Streams and Filters

## The RoboCall App

RoboCall app is used for automating the communication with groups of people.

- It can contact individuals by phone, email, or regular mail.
- In the lesson examples, the app will be used to contact three groups of people.
  - Drivers: Persons over the age of 16
  - Draftees: Male persons between the ages of 18 and 25
  - Pilots (specifically commercial pilots): Persons between the ages of 23 and 65
- It uses the `Person` class and creates the master list of persons you want to contact.
  - An `ArrayList of Person` objects is used for the examples that follow.

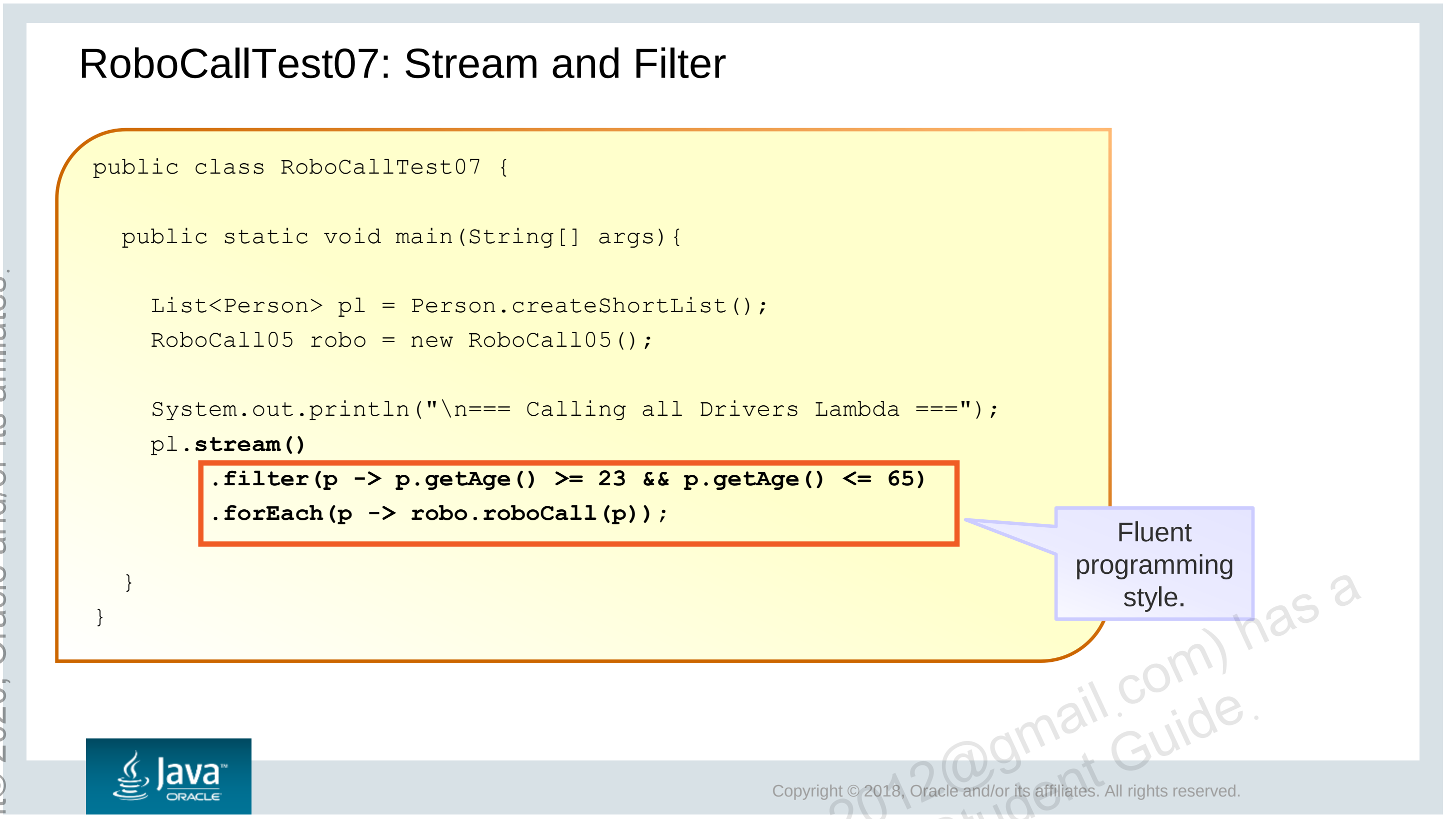
## Collection Iteration and Lambdas

Iterating with `forEach` method.

```
public class RoboCallTest06 {

  public static void main(String[] args){

    List<Person> pl = Person.createShortList();

    System.out.println("\n=== Print List ===");
    pl.forEach(p -> System.out.println(p));

  }
}
```

The `forEach` method has been added to all collections. This makes iteration much easier and provides a number of benefits. This example merely prints all the Person instances in the list.

The `Collection` interface extends the `Iterable` interface. The `Iterable` interface defines the `forEach` method.



## RoboCallTest07: Stream and Filter

```
public class RoboCallTest07 {

  public static void main(String[] args){

    List<Person> pl = Person.createShortList();
    RoboCall05 robo = new RoboCall05();

    System.out.println("\n=== Calling all Drivers Lambda ===");
    pl.stream()
        .filter(p -> p.getAge() >= 23 && p.getAge() <= 65)
        .forEach(p -> robo.roboCall(p));

  }
}
```

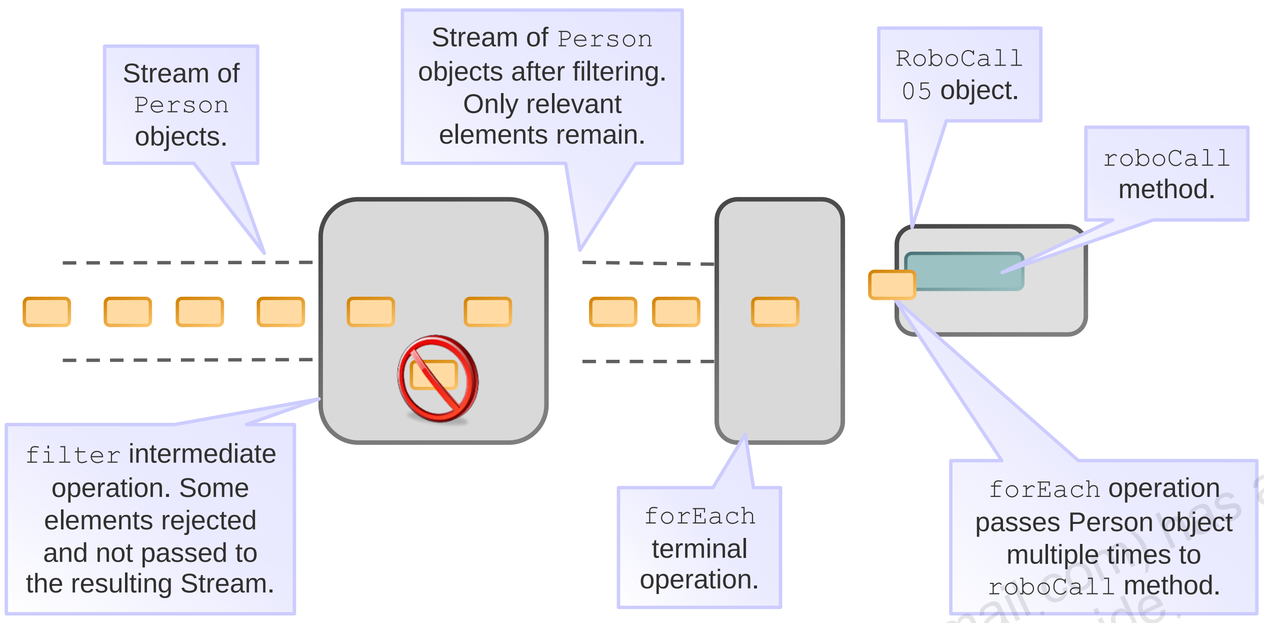
Fluent programming style.

Adding the `stream()` method to the statement opens up a whole host of new operations on collections. The `filter()` method, shown in the slide, is an example. It takes a `Predicate` as a parameter and filters the result so that only collection elements that match the `Predicate` criteria are returned to `forEach`.

Also, and this is very important, it returns a `Stream`, ensuring that further method calls can be made on the Stream by simply chaining them. This is fluent programming.

This is a big improvement on the looping shown in the previous code. First, the collection statement with a stream really describes what is happening (take this collection, filter out these elements, and return the results). Second, the looping methods in `RoboCall05` are no longer needed. The selection of elements and their output are handled in one statement.

## Stream and Filter


Stream of Person objects.
Stream of Person objects after filtering. Only relevant elements remain.
RoboCall05 object.
roboCall method.
filter intermediate operation. Some elements rejected and not passed to the resulting Stream.
forEach terminal operation.
forEach operation passes Person object multiple times to roboCall method.

## RobocallTest08: Stream and Filter Again

```
public class RoboCallTest08 {
  public static void main(String[] args){
    List<Person> pl = Person.createShortList();
    RoboCall05 robo = new RoboCall05();

    // Predicates
    Predicate<Person> allPilots =
        p -> p.getAge() >= 23 && p.getAge() <= 65;

    System.out.println("\n=== Calling all Drivers Variable ===");
       pl.stream()
          .filter(allPilots)
          .forEach(p -> robo.roboCall(p));
  }
```

This once again shows that a lambda expression can be stored in a variable and used or reused later. The chain of method calls on the stream shows a fluent programming style just as when lambda was used directly.

## `SalesTxn` Class

- Class used in examples and practices to follow
- Stores information about sales transactions
  - Seller and buyer
  - Product quantity and price
- Implemented with a Builder class
- Buyer class
  - Simple class to represent buyers and their volume discount level
- Helper enumerations
  - BuyerClass: Defines volume discount levels
  - State: Lists the states where transactions take place
  - TaxRate: Lists the sales tax rates for different states

The `SalesTxn` class is an additional data set used in the course. Each object includes data about a sales transaction. Some of the data included for each transaction includes:

- The name of the sales person who sold the product and the buyer
- The transaction date
- The number of units sold
- The price per unit
- Sales tax rates
- Any discounts applied to the transaction

## Java Streams

- Streams
  - `java.util.stream`
  - A sequence of elements on which various methods can be chained
- Method chaining
  - Multiple methods can be called in one statement
- Stream characteristics
  - They are immutable.
  - After the elements are consumed, they are no longer available from the stream.
  - A chain of operations can occur only once on a particular stream (a pipeline).
  - They can be sequential (default) or parallel.

Streams are a new type of object added to Java SE 8. A stream is a sequence of elements supporting sequential and parallel aggregate operations. These operations can be called consecutively in a single statement. This feature is called “method chaining.”

Collections and streams, while being similar, have different goals. Collections are concerned with the efficient management of and access to their elements. By contrast, streams do not provide a means to directly access or manipulate their elements. Instead, streams are concerned with declaratively describing their source and the computational operations that will be performed in aggregate on that source.

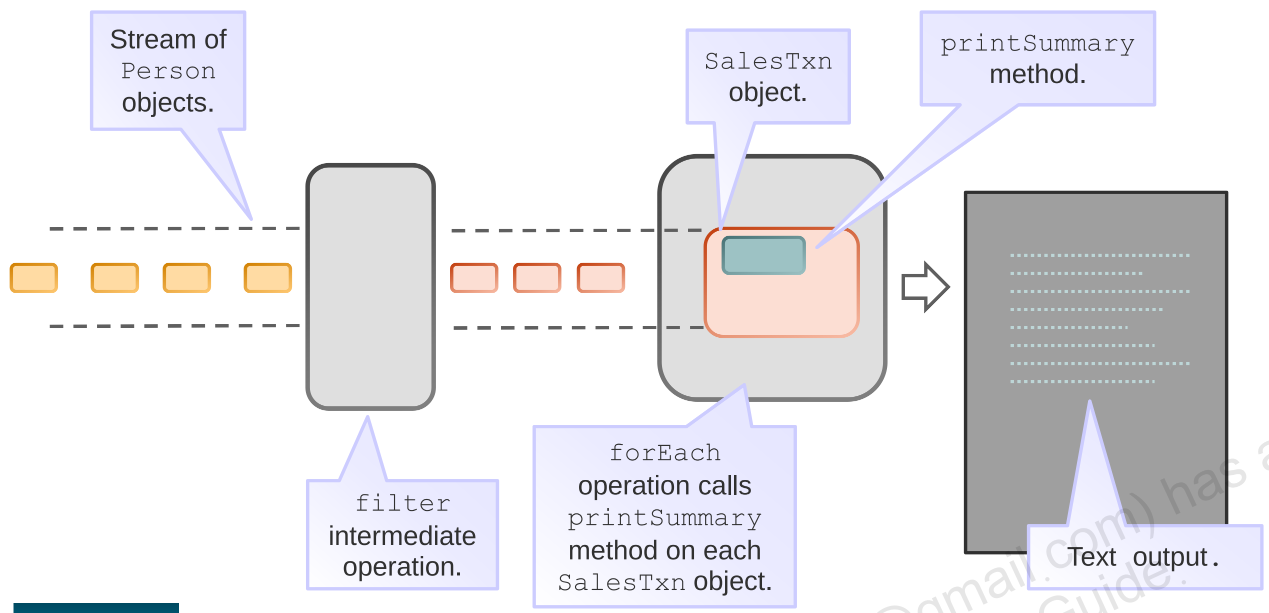
## The Filter Method

- The `Stream` class converts a collection to a pipeline:
  - Immutable data
  - Can only be used once and discarded
- The `filter` method uses a `Predicate` object (usually written as a lambda expression) to select items.
- Syntax:

```
System.out.println("\n== CA Transations Lambda ==");
tList.stream()
    .filter(t -> t.getState().equals("CA"))
    .forEach(SalesTxn::printSummary);
```

The `filter` method takes a lambda expression as a parameter (actually a Predicate object) and filters the data based on the logical expression provided. The `Predicate` object is the target type of the filter. The filter method returns a new Stream, and the `forEach` method is then called on this Stream. The forEach method calls the printSummary method of SalesTxn on each element of the Stream.

## Filter and SalesTxn Method Call


Stream of Person objects.
SalesTxn object.
printSummary method.
filter intermediate operation.
forEach operation calls printSummary method on each SalesTxn object.
Text output.

## Method References

- In some cases, the lambda expression merely calls a class method.
  - `.forEach(t -> t.printSummary())`
- In these cases, you can use a method reference.
  - `.forEach(SalesTxn::printSummary));`
- You can use a method reference in the following situations:
  - Reference to a static method
    - `ContainingClass::staticMethodName`
  - Reference to an instance method
  - Reference to an instance method of an arbitrary object of a particular type (for example, `String::compareToIgnoreCase`)
  - Reference to a constructor
    - `ClassName::new`

In many situations, a method reference can be substituted for a lambda expression. If a lambda expression merely calls a method on that object, a Method Reference can be substituted.

## Method Chaining

- Pipelines allow method chaining (like a builder).
- Methods include filter and many others.
- For example:

```
tList.stream()
  .filter(t -> t.getState().equals("CA"))
  .filter(t -> t.getBuyer().getName().equals("Acme Electronics"))
  .forEach(SalesTxn::printSummary);
```

Developers are not limited to only one method call. Multiple methods can be chained together in a single statement. This is called "method chaining." The statement structure looks similar to that of the builder pattern, and it shares the approach that each method call returns an object on which further method calls can be made.

**Analogy:** If you think about it, these statements are very similar to SQL statements with `where` clauses. The syntax is different, but the idea is very similar; elements are not addressed individually; instead, the processing is aggregate.

## Method Chaining

- You can use compound logical statements.
- You select what is best for the situation.

```
System.out.println("\n== CA Transations for ACME ==");
tList.stream()
  .filter(t -> t.getState().equals("CA") &&
      t.getBuyer().getName().equals("Acme Electronics"))
  .forEach(SalesTxn::printSummary);

tList.stream()
   .filter(t -> t.getState().equals("CA"))
   .filter(t -> t.getBuyer().getName().equals("Acme Electronics"))
   .forEach(SalesTxn::printSummary);
```

Compound logical expression.

Method chaining.

Sometimes method chaining can be an aesthetic choice over a compound logical expression.

## Pipeline Defined

- A stream pipeline consists of:
  - A source
  - Zero or more intermediate operations
  - One terminal operation
- Examples
  - Source: A Collection (could be a file, a stream, and so on)
  - Intermediate: `filter`, `map`
  - Terminal: `forEach`

In this lesson, streams have only been filtered and the results printed out. However, many more Stream methods can be chained, each one returning a Stream to be used by the next method call. Connecting these Stream operations together in a single statement is called a stream pipeline.

Note that while many Stream methods return a Stream of the same type, a method like map can return a Stream of a different type, and some methods return non-Stream types. For example, some methods return an Optional (though this can then be used to generate a Stream if that is required).

A stream pipeline consists of a source (which might be an array, a collection, a generator function, an I/O channel, etc.), zero or more intermediate operations (which transform a stream into another stream, such as filter(Predicate)), and a terminal operation (which produces a result or side-effect, such as count or forEach method). Streams are lazy; computation on the source data is performed only when the terminal operation is initiated, and source elements are consumed only as needed.

## Summary

In this lesson, you should have learned how to:

- Describe the Builder pattern
- Iterate through a collection by using lambda syntax
- Describe the Stream interface
- Filter a collection by using lambda expressions
- Call an existing method by using a method reference
- Chain multiple methods together
- Define pipelines in terms of lambdas and collections

## Practice 7: Overview

This practice covers the following topics:

- Practice 7-1: Update RoboCall to use Streams
- Practice 7-2: Mail Sales Executives using Method Chaining
- Practice 7-3: Mail Sales Employees over 50 using Method Chaining
- Practice 7-4: Mail Male Engineering Employees under 65 using Method Chaining