

Java 8

Lambda Expressions and the Stream API

Joe DiFebo

What's new in Java 8?

- Lambda Expressions
- Stream API
- The `Optional` data type
- Security Enhancements
- JavaFX Improvements
- New and Improved Tools
- ... and much more!

What are Lambda Expressions?

- A lambda expression is an anonymous function that is typically passed as a parameter to other functions.
- While Lambda Expressions are new to Java, they have been around for decades in other languages.

Javascript

```
function(x){return x * 2} or x => x * 2
```

Ruby

```
{ |x| x * 2 }
```

Java 8

```
x -> x * 2
```

Lambda Expressions in Java 8

- Several existing interfaces have been modified to allow using Lambda Expressions
 - Now marked with `@FunctionalInterface` annotation
 - Functional interfaces have exactly one abstract method
 - Examples include `Comparator`, `Runnable`, `ActionListener`
- New interfaces created specifically for lambda expressions and streams

Sorting a List of Employees

- Goal: Write a method that takes `List<Employee> employees` and sorts the list based on the `name` attribute

Without Java 8

```
public void sortEmployeesByName(List<Employee> employees) {  
    employees.sort(new Comparator<Employee>() {  
        @Override  
        public int compare(Employee e1, Employee e2) {  
            return e1.getName().compareTo(e2.getName());  
        }  
    });  
}
```

Sorting a List of Employees

- Goal: Write a method that takes `List<Employee> employees` and sorts the list based on the `name` attribute

With Java 8

```
public void sortEmployeesByName(List<Employee> employees) {  
    employees.sort((e1, e2) -> e1.getName().compareTo(e2.getName()));  
}
```

Lambda Expression Details

`(e1, e2) -> e1.getName().compareTo(e2.getName())`

- `(e1, e2)` are the parameters, both of type `Employee`
 - We can pick whatever names we want, I could choose `(x, y)` if I wanted to
 - Type Inference is used to figure out the type that these should be, so we don't need to specify that they are of type `Employee`
 - Still strongly typed, will throw compile-time errors for mistakes
- `e1.getName().compareTo(e2.getName())` is the method body
 - No return statement needed for one-line methods

Method References

- Provides easy-to-read lambda expressions for methods that already have a name
- Can be used anywhere that a lambda expression can be used
- Refer to a static method using `ClassName::methodName`
- Refer to an object's methods with `objectName::methodName`

Sorting a List of Employees

- Goal: Use a method reference to sort a list of employees

With Java 8 Method References

```
public class LambdaExpressionExample {
    /* other methods up here*/

    public void sortEmployeesByName(List<Employee> employees) {
        employees.sort(LambdaExpressionExample::compareEmployeesByName);
    }

    private static int compareEmployeesByName(Employee e1, Employee e2) {
        return e1.getName().compareTo(e2.getName());
    }
}
```

What is the Stream API?

[A stream is a] sequence of elements supporting sequential and parallel aggregate operations.

- Stream JavaDoc

- A stream is not a data structure, similar to iterators
- A "sequence of elements" can include
 - Collections (`List` and `Set`)
 - Objects from a database
 - Lines from a file (via `BufferedReader`)
 - Arbitrary mathematical sequences like the Fibonacci sequence
 - Can be infinite!

Stream API: anyMatch()

- Returns `true` if any element in the stream matches the given condition
- Input is a function that has one parameter and returns a boolean
 - We can use a lambda expression!

Stream API: anyMatch()

- **Goal:** Write a method that takes `List<Employee> employees` and `String name` and returns `true` if any employee in the list has that name

Without Java 8

```
public boolean containsName(List<Employee> employees, String name) {  
    for (Employee employee : employees) {  
        if (employee.getName().equals(name)) {  
            return true;  
        }  
    }  
    return false;  
}
```

Stream API: anyMatch()

- **Goal:** Write a method that takes `List<Employee> employees` and `String name` and returns `true` if any employee in the list has that name

With Java 8

```
public boolean containsName(List<Employee> employees, String name) {  
    return employees.stream()  
        .anyMatch(employee -> employee.getName().equals(name));  
}
```

Stream API: `allMatch()` and `noneMatch()`

- Methods work exactly the same as `anyMatch()`
- `allMatch()` returns true if all elements in the stream satisfy the given function
- `noneMatch()` returns true if no elements in the stream satisfy the given function

Aside: The Optional Class

- Class `Optional<T>` defined in `java.util`
- Useful when a method might not return a value
- Better than returning `null` since it informs the user that they must check if the value is present
- Contains methods like `isPresent()`, `get()`, and `orElse()`

Optional: Sample Usage

- Suppose that `getName()` returns `Optional<String>`

Sample Usage

```
Optional<String> name = getName();  
if (name.isPresent()) {  
    System.out.println(name.get());  
}  
else {  
    System.out.println("No name was found!");  
}
```


Optional: Sample Usage

- Suppose that `getName()` returns `Optional<String>`

Provide a default value

```
Optional<String> name = getName();  
System.out.println(name.orElse("No name was found!"));
```

Throw an exception

```
Optional<String> name = getName();  
System.out.println(name.orElseThrow(() -> new Exception()));
```

Throw an exception with a method reference

```
Optional<String> name = getName();  
System.out.println(name.orElseThrow(Exception::new));
```

Optional: Sample Usage

- Suppose that `getEmployee()` returns `Optional<Employee>`

Sample Usage

```
Optional<Employee> employee = getEmployee();  
if (employee.isPresent()) {  
    System.out.println(employee.get().getName());  
}  
else {  
    System.out.println("No employee was found!");  
}
```

Optional: Sample Usage

- Suppose that `getEmployee()` returns `Optional<Employee>`

Better Sample Usage

[illegible]

What just happened?

- `employee.map()` returns a new `Optional` of a different type
 - `emp -> emp.getName()` means the new `Optional` will be of type `String` since `emp.getName()` returns a `String`
- Now that we have an `Optional<String>` again, we can use `.orElse("No employee was found!")` to provide a default `String` value

Stream API: max()

- Returns an `Optional` containing the maximum element of a stream
 - Will return an empty `Optional` if the stream is empty
 - Takes 1 parameter, a `Comparator` function
 - There is also a `min()` function
-
- Goal: Write a method that finds the employee with the highest salary in a list and return an `Optional<Employee>` of that employee
 - Return an empty `Optional` if the list is empty

Stream API: max()

Without Streams

```
public Optional<Employee> findHighestPaidEmployee(  
    List<Employee> employees) {  
    if (employees.size() == 0 ) {  
        return Optional.empty();  
    }  
    else {  
        Employee highestEmployee = employees.get(0);  
        for (Employee employee : employees) {  
            if (employee.getSalary() > highestEmployee.getSalary()) {  
                highestEmployee = employee;  
            }  
        }  
        return Optional.of(highestEmployee);  
    }  
}
```

Stream API: max()

- Goal: Write a method that finds the employee with the highest salary in a list and return an `Optional<Employee>` of that employee
 - Return an empty `Optional` if the list is empty

Without Streams

```
public Optional<Employee> findHighestPaidEmployee(  
    List<Employee> employees) {  
  
    return employees.stream()  
        .max((e1, e2) -> Integer.compare(e1.getSalary(), e2.getSalary()));  
}
```

Terminal vs Intermediate Operations

- Terminal operations return a useful value
 - `anyMatch()` returns a `boolean`
 - `max()` returns an `Optional`
- Intermediate operations return a new stream as a result
 - Does not modify the source of the stream (the underlying list for example)
 - Can be chained together into a pipeline to perform several operations
 - Use lazy evaluation; no work will be done until a terminal operation is called
 - Examples include `filter()`, `map()`, `sorted()`, `limit()`, `distinct()`

Stream API: `map()` and `collect()`

- `map()` creates a new stream by applying a function to each element of an existing stream
- `collect()` "combines" elements of a stream in some way
 - Usually used for putting elements into a new collection
 - Convenience classes can be used via the `Collectors` class
 - i.e. `.collect(Collectors.toList())`
 - Alternatively can specify your own functions for more precise behavior
 - This is a terminal operation since it returns a useful object and not a stream

Stream API: map() and collect()

- **Goal:** Write a method that takes `List<Employee> employees` and returns `List<String>` of the employees' names

Without Java 8

```
public List<String> extractNames(List<Employee> employees) {  
    List<String> names = new ArrayList<String>(employees.size());  
    for (Employee employee : employees) {  
        names.add(employee.getName());  
    }  
    return names;  
}
```

Stream API: map() and collect()

- **Goal:** Write a method that takes `List<Employee> employees` and returns `List<String>` of the employees' names

Without Java 8

```
public List<String> extractNames(List<Employee> employees) {  
    return employees.stream()  
        .map(emp -> emp.getName())  
        .collect(Collectors.toList());  
}
```

Stream API: `filter()` and `count()`

- `filter()` creates a new stream by removing some elements from the original stream
 - Takes a function that returns a boolean, just like `anyMatch()`
- `count()` simply returns the number of elements in the stream
 - Returns a `long`, just in case the stream is huge!

Stream API: filter() and count()

- Goal: Write a method that takes `List<Employee> employees` and returns the number of employees whose office is equal to "Ann Arbor"

Without Java 8

```
public long countAnnArbor(List<Employee> employees) {  
    long count = 0;  
    for (Employee employee : employees) {  
        if (employee.getOffice().equals("Ann Arbor")) {  
            count++;  
        }  
    }  
    return count;  
}
```

Stream API: filter() and count()

- Goal: Write a method that takes `List<Employee> employees` and returns the number of employees whose office is equal to "Ann Arbor"

With Java 8

```
public long countAnnArbor(List<Employee> employees) {  
    return employees.stream()  
        .filter(emp -> emp.getOffice().equals("Ann Arbor"))  
        .count();  
}
```

Stream API: `distinct()`

- `distinct()` creates a new stream by removing duplicate elements from the original stream
 - Takes no parameters
 - Uses `.equals()` to check for equality
- Goal: Write a method that takes `List<Employee> employees` and returns the number of different offices there are among the employees
 - Hint: Use `map()` and `count()` as well

Stream API: distinct()

- Goal: Write a method that takes `List<Employee> employees` and returns the number of different offices there are among the employees

With Java 8

```
public long countNumberOfOffices(List<Employee> employees) {  
    return employees.stream()  
        .map(emp -> emp.getOffice())  
        .distinct()  
        .count();  
}
```


Stream API: `findFirst()` and `findAny()`

- `findFirst()` returns an `Optional` containing the first element in the stream
 - Returns an empty `Optional` if the stream has no elements
- `findAny()` returns an `Optional` containing an element in the stream
 - Not guaranteed to be the first element
 - Might be faster when processing streams in parallel
- Usually want to perform a `filter()` first

Stream API: findAny()

- **Goal:** Write a method that takes `List<Employee> employees` and `String email` and returns an `Optional` of that employee
 - We can assume that each employee has a unique email

With Java 8

```
public Optional<Employee> findByEmail(  
    List<Employee> employees, String email) {  
    return employees.stream()  
        .filter(emp -> emp.getEmail().equals(email))  
        .findAny();  
}
```

Stream API: `sorted()` and `limit()`

- `sorted()` creates a new stream by sorting the original stream
 - One version takes no parameters, uses natural ordering
 - Second version takes a `Comparator` just like `max()`
- `limit()` truncates the stream to be no longer than a given size
 - Takes a `long` as a parameter
 - If the stream isn't that long, the entire stream is returned

Stream API: sorted() and limit()

- Goal: Write a method that returns the 10 highest paid employees in Ann Arbor

With Java 8

```
public List<Employee> tenHighestPaidAnnArbor(List<Employee> employees) {  
    return employees.stream()  
        .filter(emp -> emp.getOffice().equals("Ann Arbor"))  
        .sorted((e1, e2) -> Integer.compare(e2.getSalary(), e1.getSalary()))  
        .limit(10)  
        .collect(Collectors.toList());  
}
```

Stream API: mapToInt()

- `mapToInt()` returns an `IntStream`, a special type of stream to deal with `int` primitives
 - Takes a function that maps to an `int`, e.g. `emp -> emp.getSalary()`
 - Has `average()` and `sum()` convenience methods that don't work on arbitrary objects
- `mapToDouble()` and `mapToLong()` also exist for those primitives

Stream API: mapToInt()

- Goal: Write a method that returns the average employee salary
 - Return 0 if the list is empty

With Java 8

```
public double findAverageSalary(List<Employee> employees) {  
    return employees.stream()  
        .mapToInt(emp -> emp.getSalary())  
        .average() // this returns an OptionalDouble!  
        .orElse(0);  
}
```

Important Notes and Fun Facts

- To preserve correct behavior, two rules must be followed
 1. Streams must be non-interfering (they do not modify the stream source)
 2. Must be stateless (results should not depend on any state that might change during execution)
- Streams cannot be reused after a terminal operation is invoked
 - Remember, no work is done until a terminal operation is used
- In some cases, streams can be infinite
 - Many methods will never return for infinite streams

Recap of Stream Methods

- Intermediate Operations

- `map()`
- `filter()`
- `distinct()`
- `sorted()`
- `limit()`
- `mapToInt()`,
`mapToDouble()`,
`mapToLong()`

- Terminal Operations

- `anyMatch()`, `allMatch()`,
`noneMatch()`
- `max()`, `min()`
- `collect()`
- `count()`
- `findAny()`

Other Important Methods

- `reduce()`
 - Extremely flexible, can be used to implement several terminal operations
 - Rarely needed in practice
- `collect()` (the other method signature)
 - Useful for loading data into arbitrary data structures
 - Most use cases are already covered by the `Collectors` class

Other Important Methods

- `toArray()`
 - Excellent if legacy code expects an array and not a list, use it if you need to
- `forEach()`
 - Also extremely flexible, lets the programmer execute arbitrary code for each element in a stream
 - Very easy to violate stream contract and potentially get unexpected behavior
 - Can just write a for loop instead

Parallel Processing

- Many streams implement the `.parallel()` method
- Automatically enables parallel processing of the stream
 - Work is divided between multiple threads
 - After threads complete, end result is then merged together
- Can actually be less efficient for small streams with simple operations
 - Millions of elements is still "small".
- Can potentially be much faster for very large streams or when the operations involved are time consuming

Parallel Processing

- As with all parallel processing, side-effects must be carefully accounted for
 - Two threads modifying the same variable at the same time will cause errors
- Side-effects are highly discouraged even for sequential streams

Parallel Processing Gone Wrong

- What is the output of the following code?

100000, 47270, 46942, 65382, or 40942?

Bad Parallel Processing

```
static int n = 0;
public static void main(String[] args) {
    IntStream.range(0, 100000).parallel().forEach(i -> n++);
    System.out.println(n);
}
```

Summary

- Streams can perform useful operations on collections
- Intermediate operations return new streams based on modifying the elements of the previous stream
- Terminal operations return useful values
- Stream operations can take lambda expressions to shorten code
- Many streams support parallel execution
 - Must be extra careful to ensure correct behavior

Thank You!

Questions?