LAS3006 – Extreme Java Development

LECTURE 2 – INTERFACES AND STREAMS

Default Methods

- Up till Java 7, interfaces were restricted to only contain **abstract methods** (i.e. methods without an implementation).
- This has been an inconvenience in certain program designs that need their interfaces to evolve or maximise reuse:
  - The library with classes implementing an interface cannot be updated.
  - The implementation is the same across most classes, but the traditional approach of extending a common super class cannot be used (cross-cutting concerns).
- The designers of Java 8 actually faced this issue with the Collections library, where interfaces needed to be extended, but would have broken most of the existent code bases (violating the backward compatibility guarantee.)
- For this reason **default methods** were introduced to interfaces, effectively adding new methods but also providing a default implementation.
- Interfaces with default methods behave similar to **traits** in other programming languages (such as Scala).

Have a look at the difference between the Java 7 Comparator and the Java 8 Comparator!

Interfaces with Default Methods

- A **default method** can be added to an interface with the **`default`** keyword.
- An interface **extending** another interface can *optionally* **override** a default method with another default method (of the same signature).
- A class **implementing** an interface can *optionally* **override** a default method with its own, or reuse the default one.

```java
public interface Greeter {
  String reply(String greeting);

  default String greet() {
    return "Hello!";
  }
}
```

```java
public class GreeterImpl implements Greeter {
  @Override
  public String reply(String greeting)
  {
    return greeting + " to you too";
  }
}
```

```java
public class ItalianGreeterImpl implements Greeter {
  @Override
  public String reply(String greeting)
  {
    return greeting + " anche a te.";
  }

  @Override
  public String greet()
  {
    return "Ciao!";
  }
}
```

Caveats with Default Methods

- Default methods provide an elegant way to extend interfaces and support backward compatibility, but they have to be used with caution.
- They introduce the problem of multiple inheritance, present in other programming languages such as C++, which Java has always avoided by enforcing single inheritance.
- They allow you to ***pollute*** your interfaces with code. Interfaces were intended to be the contracts between two modules abstracting from the implementation details.
- With default methods, interfaces start to behave like abstract classes, but not quite.

Handling Multiple Inheritance

- If a class implements two interfaces, with both providing a **default method** with **the same signature**, it will result in a **compilation error**.
- This is known as the Diamond Problem, and refers to the ambiguity of extending more than one class with one or more identical method signatures.
- In this situation, the implementing class must provide an explicit implementation of the method, which will take **precedence**.
- In the explicit implementation, the class can still call the implementation of the interface it prefers to use, using the **`super`** keyword.

```java
public interface Greeter {
  String reply(String greeting);

  default String greet() {
    return "Hello!";
  }
}
```

```java
public class GreeterImpl
      implements Greeter, Dialog {
  @Override
  public String reply(String greeting) {
    return greeting + " to you too";
  }

  @Override
  public String greet() {
    return Greeter.super.greet();
  }
}
```

```java
public interface Dialog {
  String reply(String greeting);

  default String greet() {
   return "Welcome";
  }
}
```

Interface Static Methods

- Java 8 now allows you to add **static methods** to interfaces.
- Up to Java 7, they were not allowed to keep interfaces pure contracts (and static methods were placed elsewhere such as utility classes).
- Static methods provide an alternative way to support utility methods without having to create a separate class. This is especially useful for factory methods.

```java
public interface Locator {
  String getAddress();

  static String getSystemCountry() {
    return Locale.getDefault().getCountry();
  }
}
```

Remember that you can also have static constants in interfaces. However, they are implicitly final (they cannot be modified from a static interface method).

Exercise

Continue on the previous students exercise.

1. Create a new interface **`Student`** with 3 abstract methods **`getName()`**, **`getSurname()`** and **`getId()`**. Add a default method **`getNationality()`** that returns *"Maltese"*. Make **`StudentData`** implement **`Student`**. Modify its **`toString()`** to also return the nationality.

2. Add a new class **`InternationalStudentData`** that extends **`StudentData`** but expects the nationality in the constructor (and overrides the default method in the interface). Add some international students to the students array.

3. Create a new interface **`Staff`** with the same abstract and default methods of **`Student`** and an abstract method **`getNiNumber()`**. Create a new class **`StaffData`** which implements **`Staff`**. Create a new array of staff people and print them out.

4. The university decided to allow **local `Staff`** to also be students! Create a new class **`StaffStudent`** that implements both **`Student`** and **`Staff`** interfaces. The **`Staff`** interface should take precedence if you encounter any conflicts. Test your program by creating an array of students that are also staff.

Interface Private Methods

- **`default`** and **`static`** methods introduced in Java 8 always needed to be public, which meant implementation reuse had to be exposed.
- Java 9 introduces **`private`** interface methods, allowing you to hide implementation details that you might want to reuse in **`public`** ones.
- A **`private`** interface method cannot be overridden.

Interface Method Modifiers	Supported In
`public abstract`	JDK 1
`public` **`static`**	JDK 8
`public` **`default`**	JDK 8
`private`	JDK 9
`private static`	JDK 9

Modifiers in *italics* are implicit and can be omitted.

The Stream API

- Java 8 introduced a new abstraction mechanism for **processing collections**, called **Streams**.
- The objective is to **separate** the **logic** to be applied to elements within a collection from the **iterating and scheduling** to apply such logic.
- This separation allows you to easily change the iterating and scheduling implementations without affecting your logic.
  - For example, you may opt to change your iteration from a serial to a parallel one that uses multiple concurrent threads.
- Streams on collections, arrays, iterators etc. are different from the traditional I/O `InputStream` and `OutputStream`!
- Streams provide a **functional-style interface** to a stream of elements.

Advantages of Streams

- Cleaner code (no need to perform iterations yourself).
- Efficiency through Lazy processing (no need for intermediate collections if you need intermediate processing, you can just compose streams together instead).
- Easier to change implementation to parallelize processing.

***Example:** From a Collection of **Persons**, get the ones whose age is 18 or greater and store them in a List.*

```
List<Person> over18 = new ArrayList<>();
for (Person person : persons) {
  if (person.getAge() >= 18) {
    over18.add(person);
  }
}
```

Using traditional collection processing with external iteration

```
List<Person> over18 = persons.stream().filter(person -> person.getAge() >= 18)
                                      .collect(Collectors.toList());
```

Using stream processing with internal iteration

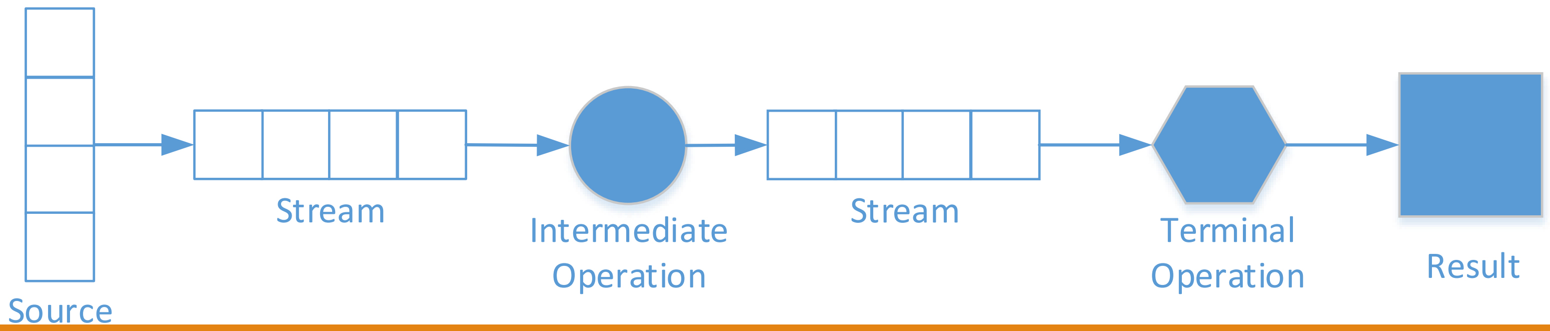
Characteristics of Streams

- **Streams do not mutate their source.**
  - They just apply the necessary logic to produce the required result.
- **Streams do not store any elements** (unless they absolutely have to)**.**
  - They are just abstract views on other data stores (such as collections) or generators (instantiating new elements on demand).
- **Stream operations are *lazy*** (when possible).
  - They do not apply any logic until their result is actually required. This also means that you can have *infinite streams*.

Streams let you focus on ***what*** needs to be done on the elements rather than ***how*** *(in which order, on which thread, etc.)*

Working with Streams

- A **stream** represents a sequence of elements.
- You can obtain a stream from methods such as **`Collection.stream()`**, **`Arrays.stream()`**, **`Stream.of()`** and **`StreamSupport.stream()`**.
- You can create your own stream.
- You can do *intermediate operations* on a stream, to transform it into another.
- You can apply *terminal operations* to produce some result.
- A stream can only be used **once**!



Stream Creation

- There are various mechanisms to create a stream from an existent collection or array of objects.
  - The `stream()` method was added to the `Collection` interface.
  - A set of static `stream()` methods were added to the `Arrays` class.
  - `Stream.of()` can also be used on a single element, array or, varargs.
  - `StreamSupport.stream()` for more advanced stream creation from a data source that has a `Spliterator` (a splittable iterator).
- You can use **generator** functions to create infinite streams. The stream will call your generator each time it needs a new value.

```
Stream<Double> randomDoubles = Stream.generate(Math::random);
```

- You can also create an empty stream via `Stream.empty()`.

Working with an Infinite Stream

- You can provide your own **generator** function that the stream calls each time it needs a new value.
- You can also have the stream pass a ***seed*** value each time, to keep track of what the last element of the sequence was. In this case, the function is called an **iterator**. The first element of the stream will be the seed itself.
- Infinite streams need to be managed more carefully when it comes to consuming them. You can use the **`limit(n)`** method, to return a finite stream of **n** elements or less from another stream.
- Once you have a **finite** stream you can either collect it into a collection, or do something with its elements directly with **`forEach()`**.

```
//geometric sequence with seed 1
Stream.iterate(1, number -> number * 2)
      .limit(10)
      .forEach(number -> System.out.print(number + " "));
```

Exercise

- Write an infinite stream which uses a **generator** function to provide all the multiples of 3, starting from 3

Hint: You have to use **`Stream.iterate()`**

- Limit it to 10 elements.
- For each element in the (finite) stream print it on screen, followed by a space.

Generating a Finite Stream

- The **`generate()`** and **`iterate()`** methods were designed for infinite streams.
  - There is no simple way to tell the stream there are no more elements.
- Java 9 added a new **`iterate()`** method, which allows you to specify when the stream has terminated.
- This syntax has the same expressive power of a **`for`** loop.

```
Stream.iterate(1, number -> number <= 512, number -> number * 2)
      .forEach(number -> System.out.print(number + " "));
```

“Zero or One”-element Streams

- It is often necessary to compose a Stream of ***zero or one*** elements.
  - Useful when you have an element which could be **`null`**.
- Java 9 introduced the **`Stream.ofNullable()`** method which returns:
  - A stream of one element if the argument is non-null
  - An empty stream if the argument is null.

```
Stream<String> stream = nullable == null ? Stream.empty() : Stream.of(nullable);
```

Java 8

```
Stream<String> stream = Stream.ofNullable(nullable);
```

Java 9

This method is very similar to ***`Optional.ofNullable()`***

Intermediate Operations

- The `Stream` class provides several methods to process the sequence of elements it is representing and produce another stream.
- Intermediate operations can thus be ***chained*** together.
- Most intermediate operations expect a functional interface (can be a lambda expression) to define the actual logic of how each element should be processed.
- Examples of intermediate operations include **filtering** and **mapping**.
- Intermediate operations can also be ***stateful***.

Filtering

- The **`Stream.filter()`** method uses a (built-in) functional interface called **`Predicate`**.
- The abstract method of the functional interface **`Predicate`** just takes an element and returns a **`boolean`** (true or false).
- The **`Stream.filter()`** method returns a new **`Stream`** without the elements that do not satisfy the **`Predicate`**.
- You can apply further operations on the returned **`Stream`**, such as filtering with another criteria.

```
String[] strings = { "Hello", "Goodbye", "Hi", "Lo" };

//filter in strings that are up to 3 characters long
Stream<String> streamShortStrings = Arrays.stream(strings).filter(s -> s.length() <= 3);

//filter in those that begin with "H"
Stream<String> streamShortNoH = streamShortStrings.filter(s -> s.startsWith("H"));
```

Exercise

- Using your previous Students exercise, add a new **enum** called **`Gender`** and add at least **`MALE`** and **`FEMALE`** to it (feel free to add others).
- Modify your **`StudentData`** class by adding a new field gender (of type Gender) and initialise it through the Constructor, add a getter method and update the **`toString()`** accordingly. Update your array.
- From your students array, create a new **`Stream`** of only **MALE** students and another stream of **non-MALE** students. (Hint: Use the methods **`Stream.of()`** and **`filter()`**).
- Consume both streams, by first printing out the MALE students and then the non-MALE ones.

Mapping Streams

- Apart from filtering a stream, we can also map it into a completely different stream, possibly of a different type.
- The **`Stream.map()`** method transforms your stream into another one, through some function that you provide to perform the transformation.
- The function you provide is applied to each element as the stream is consumed, with the return value of your function fed into the new stream.
- The **`map()`** method is an **intermediate operation** so it can be chained with other stream operations.

```
String[] strings = { "Hello", "Goodbye", "Hi", "Lo"};

//stream the length of each string
Stream<Integer> stringLengths = Stream.of(strings).map(String::length);
```

Exercise

- The Fibonacci sequence is seeded by 2 numbers, 0 and 1. The formula to generate the sequence is $x_n = x_{n-1} + x_{n-2}$.
- The resultant sequence looks like this: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

1. Write a class **`FibPair`** which has two (final) member integers, **`prev`** and **`next`**, initialised through constructor arguments.
2. Add a method **`generate()`** which returns a new instance of **`FibPair`** which is the successor of the current pair (i.e. prev is assigned to the current next, and next is assigned to the current prev + current next).
3. In your **`main()`** method use a stream of **`FibPair`** objects to generate the Fibonacci sequence, and map it to integers to get the final sequence. Print the first 10 numbers of the sequence separated by space.

Mapping and Flattening

- Sometimes you need to transform each element in a stream into many corresponding elements (one-to-many).
  - Using just **`map()`** will give you a stream of collections.
- However, sometimes you need a contiguous stream of elements.
- The **`Stream.flatMap()`** function works just like **`map()`** but also flattens the results into one stream of elements.
- The mapping function you provide must return a **`Stream`**.

```
List<User> users = …

//The method List<User> User.getFriends() returns a list of users who are friends
Stream<User> friendsOfUsers = users.stream().flatMap(user -> user.getFriends().stream());
```

Exercise

Each student can provide a list of email addresses, on which he may be notified about the courses he is attending.

1. Using the previous Students exercise, add a list of email addresses to the `StudentData` class. (You can use an array or a list of `String`).
   - Populate the students' array with a few email addresses for each student.
2. You need to notify **all** the students on **all** the email addresses they provided. Create a stream of email addresses from all the students using `flatMap()`, making no distinction between who provided the email address.
3. Go through all the elements in the email addresses stream and print them.

Substreams and Concatenation

- Streams provide the facilities to extract part of a stream into another substream.
- The **`limit(n)`** method returns a new stream that ends after **n** elements (or before if the stream has less than **n** elements).
- The **`skip(n)`** method returns a new stream that discards the first **n** elements from the original stream. If the original stream contains **n** elements or less, the returned stream will be empty.
- You can also concatenate two streams together with the **static** method **`Stream.concat()`**. Be careful when using concatenated streams:
  - The first stream has to be *finite*, otherwise the second stream is never processed.
  - It might be tempting to merge lots of streams together. However constructing lots of layers of concatenated streams can lead to performance issues.

Conditional Substreams

- If you want to process all the elements in a **`Stream`** until/from an element which matches a condition, Java 9 makes it much easier:
  - The **`takeWhile()`** method returns a new stream that ends when an element which does not match the specified **`Predicate`** is found.
  - The **`dropWhile()`** method returns a new stream that starts when an element which does not match the specified **`Predicate`** is found.

```
Stream.iterate(1, n -> n * 2)
      .takeWhile(n -> n < 50)
      .forEach(System.out::println);
```

```
1
2
4
8
16
32
```

```
Stream.iterate(1, n -> n < 50, n -> n * 2)
      .dropWhile(n -> n < 10)
      .forEach(System.out::println);
```

```
16
32
```

Sorting

- The **`Stream`** class provides the **`sorted()`** method to sort its elements.
- Obviously it must *not be an infinite stream*. It has to process all the elements to determine which is the smallest (or largest).
- If used on its own, **`sorted()`** uses the natural ordering of the elements (assuming they are comparable).
- You can also provide your own **`Comparator`** just as you do when sorting collections.
- Note that while **`Collections.sort()`** sorts collections in place (i.e. no extra memory is required) while **`Stream.sorted()`** returns a new stream which contains some data structure determining the ordering.

```
Integer[] ints = {1, 5, 2, 9, 13, 6};
Stream<Integer> intStream = Stream.of(ints).sorted();

Stream<Integer> reverseIntStream = Stream.of(ints).sorted((n1, n2) -> n2 - n1);
```

Removing Duplicates

- You can also remove duplicates from a stream.
- The **`Stream.distinct()`** method returns a new stream that keeps track of which elements were already processed, such that any duplicates are suppressed.
- The elements are returned in the same order of the original stream.
- Since it is an **intermediate operation**, it can also be chained with other stream operations (such as sorting).

```
Integer[] ints = {1, 5, 2, 5, 7, 13, 9, 13, 6};

Stream<Integer> intStream = Stream.of(ints).distinct();
```

Stateless vs Stateful Operations

- The `filter()`, `map()` and `flatMap()` methods are **stateless** intermediate operations. This means that the elements obtained from the result of such a transformation does not depend on previous elements.
- On the other hand a **stateful** operation keeps track of the elements processed through the stream, and might potentially also need to process some or all of them before returning the first element.
  - For example, a `sorted()` stream needs to process all the elements before returning the first one in order.
- When using stateful operations, you must be mindful of the potential size of the stream and the impact of such an operation on internal processing and buffering the transformation needs to do.
- **Short-circuiting** operations are those which, given an infinite stream produce a finite stream as a result. `limit()` is a short-circuiting operation.

*For more details about stream operations read Java's description of the **stream** package.*

Some Pitfalls when using Streams

1. Accidentally reusing a stream.
   - Once you consume a stream, with a collector or a **`forEach()`** you cannot use it again. If you do you will get an **`IllegalStateException`**.
2. Performing bulk operations on infinite streams.
   - Remember to use limit on infinite streams before performing bulk operations. If you perform a **`forEach()`** on an infinite stream, it will loop indefinitely.
3. Wrong order of stream operations.
   - For example, if you want 10 distinct numbers, first you have to do **`distinct()`** and then **`limit(10)`**, not the other way round.
4. Modifying the backing collection of a stream.
   - You're just inviting in race conditions, ambiguous behaviour (for instance when performing **`distinct()`** or **`sorted()`**) and the infamous **`ConcurrentModificationException`**

Exercise

- Write a small program which generates **5 unique numbers** from 1 to 42 (inclusive) using a **stream** with a **generator** and prints them out **sorted**.

Hint: Create a `java.util.Random` object and use the `nextInt()` method with an upper bound. Remember that the number returned will be from 0 (inclusive) to the upper bound (exclusive).

- Use the stream's `forEach()` to print each number on a separate line.

Increase the limit to 15 (and check you get 15 unique random numbers)

1. What happens if you put `limit()` before `distinct()`?
2. What happens if you put `sorted()` before `limit()`?
3. What happens if you increase the limit to 43?
4. What happens if you increase the limit to 43 and comment out `sorted()`?
5. Think… How would you have done this using Java 7?