EE3206
Java Programming and Applications

Lecture 11
Functional Programming and Stream API

Intended Learning Outcomes

- Recognize the change of Interface in Java 8
- Understand Functional Interface
- Learn the syntax of Lambda Expression
- Learn the syntax of Method Reference
- Apply Lambda Expression/Method Reference to improve code readability and efficiency.
- Understand Stream API
- Learn to create serial and parallel stream
- Learn to create stream from a file
- Learn to process a stream
- Learn to collect result from a stream

Functional Programming (FP)

- Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.
  - **Declarative** – computation's logic is expressed without describing its control flow (e.g. SQL).
  - **Immutable** – functions have no state and side effects, hence exhibit referential transparency.
- Eliminating side effects makes it much easier to understand and predict computational behaviour.
- It also helps make code more suitable for parallel processing, which often improves application performance.
- FP becomes more popular due to the increasing demand in big data processing which is typically a concurrent model.

Java 8 Interface Changes

- Prior to Java 8, interface in java can only have abstract methods. Java 8 allows the interfaces to have concrete **default** and **static** methods.
- The reason of this change is to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces (e.g. the need of adding functional programming support to the Collection Framework)
- Default method behaves as if it is an instance method.
- Static methods in interfaces are similar to the default methods except that static methods cannot be inherited and cannot be overridden by the classes that implements these interfaces.

Java8InterfaceExample

Lambda Expression

- **Lambda expression** and **Functional Interface** are introduced in Java 8 to support Functional Programming.
- In general, a functional interface has a single functionality to exhibit. The interface only contains one abstract method, but may contains other static and default methods.
- Lambda expression provides a clear and concise way to represent one-method-interface (i.e. functional interface) using an expression.
- A lambda expression represents an anonymous function that is treated as an instance of a functional interface.
- Lambda expressions also improve the Collection libraries making it easier to iterate through, filter, and extract data from a Collection.

Instantiating Functional Interface

- Anonymous inner classes provide a way to implement classes that may occur only once in an application. For example, in a standard Swing application a number of event handlers are required for keyboard and mouse events. Rather than writing a separate event-handling class for each event, you can write something like this.

```
JButton testButton = new JButton("Test Button");
testButton.addActionListener(new ActionListener(){
@Override public void actionPerformed(ActionEvent ae){
    System.out.println("Click Detected by Anon Class");
  }
});
```

- In Java 8, the ActionListener is known as a functional interface. Interfaces like Runnable and Comparator are used in a similar manner.

```
public interface ActionListener extends EventListener {

public void actionPerformed(ActionEvent e);

}
```


Lambda Expression Syntax

- Lambda expressions address the bulkiness of anonymous inner classes by converting five lines of code into a single statement.
- A lambda expression is composed of three parts.

Argument List	Arrow Token	Body
`(int x, int y)`	`->`	`x + y`

- The body can be either a single expression/return-statement or a block of statements.
  - In the expression form, the body is simply evaluated and returned.
  - In the block form, the body is evaluated like a method body and a return statement returns control to the caller of the anonymous method.

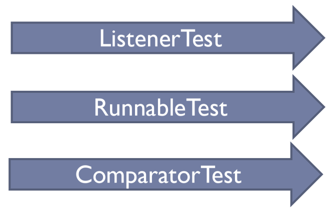
Lambda Expression Examples

- The first expression takes two integer arguments, named x and y, and uses the expression form to return x+y.
- The second expression takes no arguments and uses the expression form to return an integer 42.
- The third expression takes a string and uses the block form to print the string to the console, and returns nothing.

```
(int x, int y) -> x + y

() -> 42

(String s) -> { System.out.println(s); }
```



- The data type of the arguments is optional. Lambda supports "target typing" which infers the object type from the context in which it is used.
- When there is a single argument, if its type is inferred, it is not mandatory to use parentheses, e.g. `(a) -> return a*a;` is the same as `a -> return a*a;`

Standard Functional Interfaces

- Java 8 provides a number of standard interfaces that are designed as a starter set for developers

Interface	Method	Description
Function<T,R>	R apply(T t)	A function that takes an argument of type T and returns a result of type R. Apply a function to an input value.
BiFunction<T,U,R>	R apply(T t, U u)	A function that takes 2 arguments of types T and U, and returns a result of type R.
Predicate<T>	boolean test(T t)	A predicate is a Boolean-valued function that takes an argument and returns true or false. Test the predicate with an input value.
BiPredicate<T,U>	boolean test(T t, U u)	A predicate with 2 arguments.

Standard Functional Interfaces

Interface name	Method	Description
Consumer<T>	void accept(T t)	An operation that takes an argument, operates on it to produce some side effects, and returns no result. The consumer accepts an input item.
BiConsumer<T,U>	void accept(T t, U u)	An operation that takes 2 arguments, operates on them to produce some side effects, and returns no result.
Supplier<T>	T get()	Represents a supplier that returns a value of type T. Get an item from supplier.
UnaryOperator<T>	T apply(T t)	Inherits from Function<T,T>.
BinaryOperator<T>	T apply(T t1, T t2)	Inherits from BiFunction<T,T,T>

Using the Function<T, R> Interface

- Functional interfaces are used in 2 contexts:
  - Library designers that implement the APIs (e.g. Collection and Stream API)
  - Library users that use the APIs

FunctionUtilTest

- Example: convert centigrade to Fahrenheit

```
// Function<T, R> R apply(T t)
// T : Double
// R : Double
// argument t is called x in this example
Function<Double, Double>
   centigradeToFahrenheit = x -> (x * 9 / 5) + 32.0;

double degreeC = 36.9;
double degreeF = centigradeToFahrenheit.apply(degreeC);
```


Using the Function<T, R> Interface

- Example: calculate the aggregated quantity of all trades

```
// Function<T, R> R apply(T t)
// T : List<Trade>
// R : Integer
// argument t is called trades in this example
Function<List<Trade>, Integer> aggregatedQty =
   trades -> {
      int total = 0;
      for (Trade t : trades)
         total += t.getQuantity();
      return total;
   };


List<Trade> list = new ArrayList();
...  // statements to fill up list

int totalQty = aggregatedQty.apply(list);
```

Method Reference

- A method reference is a shorthand to create a lambda expression using an existing method.
- If a lambda expression contains a body that is an expression using a method call, you can use a method reference in place of that lambda expression.
- Types of method references (::“4 dots”)

Syntax	Description
TypeName::staticMethod	A method reference to a static method of a class, an interface, or an enum
objectRef::instanceMethod	A method reference to an instance of the specified object
ClassName::instanceMethod	A method reference to an instance method of an arbitrary object of the specified class
TypeName.super::instanceMethod	A method reference to an instance method of the supertype of a particular object
ClassName::new	A constructor reference to the constructor of the specified class
ArrayTypeName::new	An array constructor reference to the constructor of the specified array type

Using Method Reference

```
ToIntFunction<String> lenFunction = str -> str.length();

Supplier<Item> func1 = () -> new Item();

Function<String, Item> func2 = str -> new Item(str);

BiFunction<String, Double, Item>
            func3 = (name, price) -> new Item(name, price);
```

- The above lambda expressions can be rewritten using method reference.

```
ToIntFunction<String> lenFunction = String::length;

Supplier<Item> func1 = Item::new;

Function<String, Item> func2 = Item::new;

BiFunction<String, Double, Item> func3 = Item::new;
```


Optional

- In Java 8, a new class Optional<T> is used to represent a value is present or absent.
- Optional acts as a wrapper of the NULL value such that a potential nullable value is contained by a non-null Optional object that is testable.
- It can help to avoid runtime NullPointerException, and supports us in developing clean and neat Java APIs or applications.
  - Optional.of(value) : create an Optional with the given non-null value
  - Optional.empty() : create an empty Optional instance
  - isPresent() : check if the Optional has a value
  - get() : return a value from the Optional
  - orElse(T) : return the value in Optional if it is present; otherwise return T

OptionalTest

Stream API

- An ***aggregate*** operation computes a single value from a collection of values.
- A ***stream*** is a sequence of data elements supporting sequential and parallel aggregate operations.
  - https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
- Differences between streams and collections:
  - Collections focus on storage of data elements for efficient access.
  - Collections support imperative programming using external iteration.
  - Streams focus on aggregate computations on data elements from a data source that is typically, but not necessarily, collections.
  - Streams have no storage.
  - Streams can represent a sequence of infinite elements.
  - Streams are designed to support (declarative) functional programming using internal iteration.
  - Streams support lazy operations.
  - Streams can be ordered or unordered.
  - Streams are designed to be processed in parallel with no additional work from the developers.
  - Streams cannot be reused.

Iteration on Collection vs Stream

- Compute the sum of the squares of odd values in a collection using external iteration.

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

int sum = 0;
for (int n : numbers) // for (Integer n : numbers)
   if (n % 2 == 1)
   {
      int square = n * n;
      sum = sum + square;
   }
```

- Compute the sum of the squares of odd values in a stream using internal iteration.

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

int sum = numbers.stream()
                 .filter(n -> n % 2 == 1)
                 .map(n -> n * n)
                 .reduce(0, Integer::sum);
```

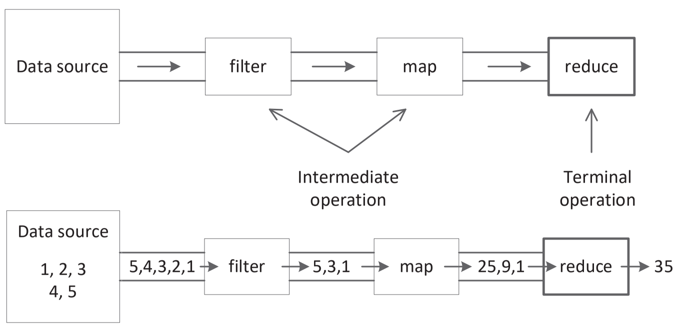

Serial Stream and Parallel Stream

- The stream() method converts a collection to a serial stream by default.
- A stream represents a sequence of elements on which various aggregate methods can be chained.
- Streams are not reusable. A stream cannot be reused after calling a terminal operation on it (e.g the reduce() method).
- If you need to perform a computation on the same elements from the same data source again, you must recreate the stream pipeline.
- In addition, streams are designed to process their elements in parallel with built-in support using the Fork/Join framework.

```
int sum = numbers.parallelStream()
                 .filter(n -> n % 2 == 1)
                 .map(n -> n * n)
                 .reduce(0, Integer::sum);
```


Intermediate and Terminal Operations

- Terminal operations are known as *eager* (or *result-bearing*) operations.
- Intermediate operations are known as *lazy* (or *non result-bearing*) operations.
- A lazy operation on a stream does not process the elements of the stream until an eager operation is called on the stream.



Debugging Stream Pipeline

- Each operation in the stream pipeline transforms the elements of the input stream either producing another stream or a result.
- Sometimes you may need to look at the elements of the streams as they pass through the pipeline.
- You can do so by using the peek(Consumer<? super T> action) method.

```
int sum = numbers.stream()
                 .filter(n -> n % 2 == 1)
                 .peek(e -> System.out.println(
                       "Filtered element: " + e))
                 .map(n -> n * n)
                 .peek(e -> System.out.println(
                       "Mapped element: " + e))
                 .reduce(0, Integer::sum);
```

Creating Stream From Values

- Using the static method Stream.of()

```
<T> Stream<T> of(T t)           // single value
<T> Stream<T> of(T... values) // multiple values
Stream<Integer> intStream = Stream.of(1,2,3,4,5);
```

- The Stream interface also supports creating a stream using the Stream.Builder<T> interface.

```
// Obtain a builder
Stream.Builder<Integer> builder = Stream.builder();
// Add elements and build the stream
Stream<Integer> intStream = builder.add(1).add(2).add(3)
                                   .add(4).add(5)
                                   .build();

// A more convenient way to build an integer stream using
// the IntStream interface
IntStream oneToFive = IntStream.range(1, 6);
// Or
IntStream oneToFive = IntStream.rangeClosed(1, 5);
```


Creating Stream From File

- We can read text from a file as a stream of strings in which each element represents one line of text from the file.
- We need a method that reads a file lazily and returns the contents as a stream of strings.
- We may use the method lines() in the java.nio.file.Files class.

```
String filename = "testdata.txt";
Path filepath = Paths.get(filename);

try (Stream<String> lines = Files.lines(filepath))
{
   lines.forEach(System.out::println);
}
catch (IOException e)
{
   e.printStackTrace();
}
```


Creating Infinite Stream

- An infinite stream is a stream with a data source capable of generating infinite number of elements.
- The Stream interface contains 2 static methods to generate an infinite stream.

```
<T> Stream<T> iterate(T seed, UnaryOperator<T> f)
// elements: seed, f(seed), f(f(seed)), f(f(f(seed))), ...
<T> Stream<T> generate(Supplier<T> s)


// Create a stream of odd natural numbers
Stream<Long> oddNaturalNum = Stream.iterate(1L, n -> n+2);

// Create a stream of the first 10 odd natural numbers
// and print it to standard output using the forEach method.
Stream.iterate(1, n -> n+2)
      .limit(10)
      .forEach(System.out::println);
```

PrimeUtilTest

Finding and Matching in Stream

- The Stream API supports different types of find and match operations on stream elements.

```
// test elements in the stream against the predicate
boolean allMatch(Predicate<? super T> predicate)
boolean anyMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)

// get an element from a stream
Optional<T> findAny()
Optional<T> findFirst()
```

FindAndMatchStream

Collecting Elements From Stream

- The collect() method of the Stream<T> interface

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R,R> combiner)

<R,A> R collect(Collector<? super T,A,R> collector)
```

- Consider the 1st version of the collect() method, where the 3 arguments are:
  - a **supplier** that supplies a container to store (collect) the results
  - an **accumulator** that accumulates the results into the container
  - a **combiner** that combines the partial results when the reduction operation takes place in parallel (i.e. using parallel stream)

Collecting Elements From Stream

- Suppose we have a steam of people, and we want to collect the names of all the people in an ArrayList<String>.

```
// Create supplier, accumulator, combiner using Lambda expression
Supplier<ArrayList<String>> supplier = () -> new ArrayList<>();
BiConsumer<ArrayList<String>, String> accumulator =
                              (list, name) -> list.add(name);
BiConsumer<ArrayList<String>, ArrayList<String>> combiner =
                              (list1, list2) -> list1.addAll(list2);

// Create supplier, accumulator, combiner using method reference
Supplier<ArrayList<String>> supplier = ArrayList::new;
BiConsumer<ArrayList<String>, String> accumulator = ArrayList::add;
BiConsumer<ArrayList<String>, ArrayList<String>> combiner =
                                    ArrayList::addAll;
```


Using Built-in Collectors

- The utility class Collectors provides out-of-box implementation for commonly used collectors.
  - https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html
  - toList(), toSet(), toMap() and toCollection() return a collector that accumulates the input elements into a new List/Set/Map/Collection.
  - joining() returns a collector that joins strings with/without a delimiter
  - counting() counts the number of elements in a stream
  - groupingBy() returns a collector that groups the data before collecting them in a Map
    - Grouping data is based on the keys returned by the key extractor (mapper) function.
  - partitioningBy() returns a collector that partition the data based on a predicate.
    - The Map returned from the collector always contains 2 entries: one with the key value as true and another with the key value as false.

CollectorTest

Collecting Summary Statistics

- In data-centric application, very often we need to compute the summary statistics on a group of numeric data.
- Java JDK provides 3 classes to collect statistics
  - java.util.DoubleSummaryStatistics
  - java.util.LongSummaryStatistics
  - java.util.IntSummaryStatistics
- Commonly used methods in the above classes
  - accept() : add a value to the data set
  - getCount()
  - getSum()
  - getMin()
  - getAverage()
  - getMax()

StatisticsTest