Application Program Development

Segment : Java Lambda’s

Mahboob Ali

Agenda for Week 6

- Lecture
  - Lambda and Streams
  - Properties and bindings
  - Generics
  - Collection
    - List, Set, Maps
  - Corresponding STL Classes
  - Java Iterators
  - Lambdas and Streams
- Lab
  - Part 1 : In-Lab – Design and create a GUI based Application
  - Part 2 : DIY – using the GUI designed in-lab to write events

Outcomes

- Understanding Lambda
  - Why the need?
  - Functional Programming
  - Imperative vs Declarative style
  - Functional Interface
  - Lambda Expression
  - Why should we care about Lambda Expression?
- Understanding Streams
  - What are they?
  - Stream API
  - Stream Operations

Why the need?

- The biggest need is due to the rise of multicore CPUs, which involves programming algorithms locks, error-prone and time-consuming.
- The `java.util.concurrent` package and the many of external libraries have developed a variety of concurrency abstractions that helps programmers to write code that performs well on multicore CPUs.
- But Java still has its limits, a good example of this is the lack of efficient parallel operations over large collections of data.
- Java 8 lambdas allows you to write complex collection-processing algorithms, and simply by changing a single method call you can efficiently execute this code on multicore CPUs.
- In order to enable writing of these kinds of bulk data parallel libraries, however, Java needed a new language change: lambda expressions.

What is *Functional Programming?*

- The feature of passing code to methods (and also being able to return it and incorporate it into data structures) also provides access to a whole range of additional techniques that are commonly referred to as *functional-style programming*.
- In a nutshell, such code, called *functions* in the functional programming community, can be passed around and combined in a way to produce powerful programming idioms.
- It is thinking about your problem domain in terms of immutable values and functions that translate between them.

Lambda

- Java Lambda is based on JSR (Java Specification Request) 335, “Lambda Expressions for the Java™ Programming Language.”
- The feature was appropriately named after lambda calculus, the formal system in mathematical logic and computer science to express computations.
- JSR 335, better known as *Project Lambda,* comprised many features, such as expressing parallel calculations on streams (the Stream API).
- A primary goal of lambdas is to help address the lack in the Java language of a good way to express **functional programming** concepts.
- Languages that support functional programming concepts have the ability to create anonymous (unnamed) functions, similar to creating objects instead of methods in Java.
- These function objects are commonly known as ***closures***.
- Some common languages that support closures or lambdas are Common Lisp, Clojure, Erlang, Haskell, Scheme, Scala, Groovy, Python, Ruby, and JavaScript.
- The main idea is that languages that support functional programming will use a closure-like syntax.
- In Java 8, you can create anonymous functions as first-class citizens.
- In other words, functions or closures can be treated like objects, so that they can be assigned to variables and passed into other functions.

Changing the Programming Thinking

- **Imperative style**—that’s what Java has provided us since its inception.
  - In this style, we tell Java every step of what we want it to do and then we watch it faithfully exercise those steps

- **Declarative style**—*what* we want rather than delve into *how* to do it.

Imperative style - Example

```
public class Cities {
  public static void findChicagoImperative(final List<String> cities)
      {
            boolean found = false;
            for(String city : cities) {
                  if(city.equals("Chicago")) {
                          found = true;
                          break;
                  }
            }
            System.out.println("Found chicago?:" + found);
      }
  public static void main(final String[] args) {
      List<String> cities = Arrays.asList("Albany", "Boulder", "Chicago",
                                          "Denver", "Eugene");
          findChicagoImperative(cities);
       }
}
```

Declarative style - Example

```
public class Cities {
  public static void findChicagoDeclarative(final List<String> cities)
   {
        System.out.println("Found chicago?:" + cities.contains("Chicago"));
   }

  public static void main(final String[] args) {
        List<String> cities = Arrays.asList("Albany", "Boulder", "Chicago",
                                                       "Denver", "Eugene");
            findChicagoDeclarative(cities);
        }
}
```

- Improvements:
  - No messing around with mutable variables.
  - Iteration steps wrapped under the hood
  - Less clutter
  - Better clarity; retains our focus
  - Less impedance; code closely trails the business intent
  - Less error prone
  - Easier to understand and maintain

JavaFX – Button onAction

```
Button btn = new Button();
btn.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
    System.out.println("Hello World");
} });
```

- You will notice that this code looks very verbose just to wire up a button.
- Buried deep in an anonymous inner class is a single line to output text. Wouldn’t it be nice to be able to express a block of code containing the behavior you want without the need of so much boilerplate code?
- Rewriting the button handler code.

```
btn.setOnAction(event -> System.out.println("Hello World") );
```

- Using lambda expressions not only makes code concise and easy to read, but the code is also likely to perform better.
- Actually, under the hood, the compiler is capable of optimizing code and likely to be able to reduce its footprint.

Lambda Expressions - Syntax

- There are two ways to specify lambda expressions.

```
(param1, param2, ...) -> expression;
(param1, param2, ...) -> { /* code statements */ };
```

- Lambda expressions begin with a list of parameters surrounded by parentheses, followed by the arrow symbol -> (a hyphen and a greater-than sign) and an expression body.
- The surrounding parentheses are *optional* only if there is one parameter defined.
- When the expression doesn't accept parameters (and is said to be *empty*), the parentheses are still required.
- Separating the parameter list and the expression body is the *arrow symbol*.
- The expression body or code block may or may not be surrounded by curly braces.
- When the expression body doesn't have surrounding curly braces, it must consist of only one statement. When an expression is a one-line statement, it is evaluated and returned to the caller implicitly.
- If the method requires a return type and your code block has curly braces, you must have a return statement.

```
// explicit return of result
Function<Double, Double> func = x -> { return x * x; }
// evaluates & implicitly returns result
Function<Double, Double> func = x -> x * x;
double y = func(2.0); // x = 4.0
```

Lambda Expression

- **Simple Method vs Lambda**

return type

parameter list

```
String doSomething(String txt){
        return txt.toUpperCase();
}
```

Function Body

- Lambda has three basic elements as well.

Parameter list

Function body

```
(String txt) -> { return txt.toUpperCase();}
```

New Arrow to indicate it is lambda

Lambda Expression

```
(String txt) -> { return txt.toUpperCase();}
```

- Lambda methods are also called <u>anonymous methods</u> due to reason they don’t have names.
- Lambda methods don’t define return type, the return type is defined in the functional interface.
- It's the context in which we assign this lambda that the compiler knows what the return type is.

Lambda Expression

```
(int x, int y) -> {
                        if (x  > y)
                              return x;
                        else
                              return y;
                  }
```

- The above lambda expression, can be used to implement the *functional Interface* whose abstract method takes two integers as arguments and returns an integer.
- Because the compiler knows the method signature of that abstract method, it isn't necessary to explicitly call out the parameter types.

```
(x, y) -> {
           if (x  > y)
                 return x;
           else
                 return y;
                 }
```

```
(x, y) -> {
             return (x > y ? x
: y);
                }
```

```
(x, y) -> (x > y ? x : y)
```

- Three Syntactically Equivalent Lambda Expressions to Set Action Code on a JavaFX Button

```
btn.setOnAction( (ActionEvent event) -> {System.out.println(event); } );

btn.setOnAction( (event) -> System.out.println(event) );

btn.setOnAction( event -> System.out.println(event) );
```

Functional Interfaces

- Any Java interface which contains one and only one abstract method.

```
@FunctionalInterface
public interface Messenger{
    void notify(String msg, int count);
}
```

- The abstract method defined in a functional interface is the contract for any lambdas which will implement the functional interface.

```
@FunctionalInterface
public interface Something<T,R>{
    R apply(T param);
}
```

- Functional interfaces can be generic interfaces or explicitly typed interfaces.

Method Reference

- Often when using lambdas, there are methods that take a single parameter as input or a single value as a return type.
- This is very redundant so, new in Java 8 is the concept of *method references*.
- **<u>Method references</u>** are basically syntactic sugar that allow you to make method calls with even less verbosity, subsequently making things easier to read.
- For example, the following is a lambda expression that uses a method reference:

```
btn.setOnAction(System.out::println);
```

- This code sets the action on the button, and you'll notice it's a concise version that behaves the same way as in before example.
- The difference is that there isn't an input parameter for the ActionEvent for the lambda and an unusual double colon between the `System.out` and the `println` method.
- The double colon is called the *scope operator*, and it references the method by name.
- You'll also notice the `println` method's parentheses are absent.
- If you remember, whenever a lambda expression takes a single parameter as input, the parameter is implicitly passed to the method println that takes a single input.
- Of course, both types must be the same.
- The event object in the example will implicitly call `toString()` to pass a String to the `println` method.

Functional Interfaces

- The advent of cloud computing has helped to reinvigorate many functional programming languages.
- It became apparent there was a paradigm shift in problem-solving that involves extremely large datasets.
- A typical use case when applying functional programming techniques is the ability to iterate over datasets while performing computations in a distributed fashion so that load can be shared among nodes or CPU cores.
- In contrast, *imperative programming languages* gather data to then be passed into a tight for loop to be processed.
- Because of how data and code are coupled, this puts a lot of the burden on one thread (core) to process so much data.
- The problem needs to be decomposed to allow other threads (cores) to participate in the computation, which then becomes distributed.
- One of the advantages of functional programming is the ability to express functionality in a syntactically concise manner, but more important is the ability to pass functionality (lambda expressions) to methods.
- Being able to pass lambda expressions to methods often fosters the concept of ***lazy evaluation***.
- This behavior is the same as function callback behavior (asynchronous message passing), where invocations are deferred (and thus “lazy”) until a later time.
- The opposite of lazy evaluation is ***eager evaluation***.
- Using lazy evaluations will often increase performance by avoiding unnecessary calculations.
- A functional interface is basically a single abstract method (SAM).
- The idea of functional interfaces has been around for a very long time.

Functional Interfaces

- For instance, those who have worked with Java threads will recall using the `Runnable interface`, where there is a single run() method with a void return type.
- The single abstract method pattern is an integral part of Java 8’s lambda expressions.

```
// functional interface
interface MyEquation {
      double compute(double val1, double val2);
}
```

- After creating a functional interface, you can declare a variable to be assigned with a lambda expression.

```
MyEquation area = (height, width) -> height * width;
MyEquation perimeter = (height, width) -> 2*height + 2*width;
System.out.println("Area = " + area.compute(3, 4));
System.out.println("Perimeter = " + perimeter.compute(3, 4));

//Output:
      Area: 12.0
      Perimeter: 14.0
```

Functional Interfaces

- **The Annotation @:**
  - The annotation has no affect on the code but just a compile time check to see if the interface under is following the *functional Interface* definition or not which is having only and only one abstract method.

- We don't require to create our own functional interfaces all the time while dealing with lambda's, mostly they are used when we delt with *Collections* and *Streams.*
- Java has a handful list of *functional interfaces* defined in `java.util.function` package. Few basic one are
  - Function: takes one argument and produces the results, used to map one object of one type to another type of object.
  - Consumer: takes one argument and doesn't return anything, used to iterate over multiple objects.
  - Predicate: take one argument and always returns true or false, used to perform key filter operations on collections of objects.

Single Parameter Lambda Syntax

```
(String txt) ->  return txt.toUpperCase()
```

- With a single parameter lambdas, not only can we drop the parameter type, but we can also lose the parentheses.
- This makes our lambda is a little lighter without losing any readability.
- The key here is that the arrow token is what signals this as a lambda expression.
- We can't forget that we can also reduce this lambda body if it contains a single statement.

Review

- No need to add parameters types: because lambdas are single abstract methods and due to functional interface, the complier knows its types.

```
(x , y) -> {...}
```

- Reduce the bodies of single statement lambdas, no need of { }, return or ;

```
(x , y) -> x * y
```

- Single parameters lambdas doesn’t require parathesis.

```
x -> {...}

() -> {...}
```

Example

```
public class Example{
	public static void main(String[] args){
		Convert<String, Boolean> str2Bool = (s) -> { return
Boolean.parseBoolean(s);};

	System.out.println(str2Bool.apply("TRUE"));
	System.out.println(str2Bool.apply("tRuE"));
	System.out.println(str2Bool.apply("faLsE"));
	System.out.println(str2Bool.apply("No"));
	System.out.println(str2Bool.apply(null));

		Convert<Boolean, Integer> Bool2Int = b -> b ? 1 : 0;

	System.out.println(Bool2Int.apply(true));
	}
}

@FunctionalInterface
interface Converter<T,R>{
	R apply(T source);
}
```

Lambda Expressions

- A *lambda expression* can be understood as a concise representation of an anonymous function that can be passed around.
- **Anonymous**— We say anonymous because it doesn’t have an explicit name like a method would normally have: less to write and think about!
- **Function**— We say function because a lambda isn’t associated with a particular class like a method is. But like a method, a lambda has a list of parameters, a body, a return type, and a possible list of exceptions that can be thrown.
- **Passed around**— A lambda expression can be passed as argument to a method or stored in a variable.
- **Concise**— You don’t need to write a lot of boilerplate like you do for anonymous classes.

Why should be care about Lambda Expression?

- In previous segment as we have seen that passing behaviors using interfaces is tedious and verbose.

- Lambda first of all fix that problem of tediousness.

- With the use of it you shouldn’t be writing clumsy code using anonymous classes.

- Your code become clearer and more flexible.

Why should be care about Lambda Expression?

- Code using Anonymous classes

```
Comparator<Apple> byWeight = new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
};
```

- Code using Lambda

```
Comparator<Apple> byWeight =
    (a1, a2) ->   a1.getWeight().compareTo(a2.getWeight());
```

`java.util.function.Predicate<T>` Interface

- Defines an abstract method named `test` that accepts an object of generic type `T` and returns a `Boolean`.

- You might want to use this interface when you need to represent a `boolean` expression that uses an object of type `T`.

- For example, you can define a lambda that accepts String objects.

`java.util.function.Consumer<T>` Interface

- Defines an abstract method named `accept` that takes an object of generic type T and returns no result (void).

- You might use this interface when you need to access an object of type T and perform some operations on it.

- For example, you can use it to create a method forEach, which takes a list of Integers and applies an operation on each element of that list.

`java.util.function.Function<T,R>` Interface

- Defines an abstract method named `apply` that takes an object of generic type T as input and returns an object of generic type R.
- You might use this interface when you need to define a lambda that maps information from an input object to an output.
- For example, extracting the weight of an apple or mapping a string to its length.

Functional Style Data Processing

- Collections is the most heavily used API in Java.
- What would you do without collections?
  - Nearly every Java application makes and processes collections.
  - Collections are fundamental to many programming tasks:
    - They let you group and process data.
    - To illustrate collections in action, imagine you want to create a collection of dishes to represent a menu and then iterate through it to sum the calories of each dish.
    - You may want to process the collection to select only low-calorie dishes for a special healthy menu.
    - But despite collections being necessary for almost any Java application, manipulating collections is far from perfect

What are Streams?

- Streams are an update to the Java API that lets you manipulate collections of data in a declarative way (you express a query rather than code an ad hoc implementation for it).

- For now you can think of them as fancy iterators over a collection of data.

- Streams can also be processed in parallel transparently, without you having to write any multithreaded code.

Streams

- A pipeline is a sequence of operations (lambda expressions/functional interfaces) that can process or interrogate each element in a stream.
- Such operations allow you to perform aggregate tasks.
- **Aggregate operations** are similar to the way spreadsheets can execute some computation over a series of cells, such as formatting, averaging, or summing up values.
- To begin using aggregate operations on collections, you will first invoke the default `stream()` method on the `java.util.Collection` interface.

```
List<Integer> values = Arrays.asList(23, 84, 74, 85, 54, 60);

Stream<Integer> stream = values.stream();
```

- The common built-in aggregate operations are `filter`, `map`, and `forEach`.
- A `filter` allows you to pass in an expression to filter elements and returns a new Stream containing the selected items.
- The map operation converts (or maps) each element to another type and returns a new `Stream` containing items of the mapped type.
- For instance, you may want to map `Integer` values to `String` values of a stream.
- A `forEach` operation allows you to pass in a lambda expression to process each element in the stream.

Streams

```
// create a list of values
List<Integer> values = Arrays.asList(23, 84, 74, 85, 54, 60);
System.out.println("values: " + values.toString());
// nonlocal variable to be used in lambda expression.
int threshold = 54;
System.out.println("Values greater than " + threshold + " converted to hex:");
Stream<Integer> stream = values.stream();
// using aggregate functions filter() and forEach()
stream
        .filter(val -> val > threshold) /* Predicate functional interface */
        .sorted()
        .map(dec -> Integer.toHexString(dec).toUpperCase() ) /* Consumer functional
                                                                  interface*/
        .forEach(val -> System.out.println(val)); /* each output values. */
```

Java 7 VS Java 8 Code

```
List<Dish> lowCaloriesDishes = new ArrayList<>();
	for(Dish d : menu){
		if(d.getCalories() < 400)
			lowCaloriesDishes.add(d);
	}


Collections.sort(lowCaloriesDishes, new Comparator<Dish>(){
	public int compare(Dish d1, Dish d2){
		return Integer.compare(d1.getCalories(), d2.getCalories())
	}
});


List<Dish> lowCaloriesDishesName = new ArrayList<>();
	for(Dish d : lowCaloriesDishes){
		lowCaloriesDishesName.add(d.getName());
	}
```

Java 7 VS Java 8 Code

```
List<String> lowCaloricDishesName =
                    menu.Stream()
                         .filter(d -> d.getCalories() < 400)
                         .sorted(comparing(Dishes::getCalories))
                         .map(Dish::getName)
                         .collect(toList());
```

- The code is written in a declarative way:
  - you specify what you want to achieve that is
    - filter dishes that are low in calories
  - As opposed to specifying
    - how to implement an operation (using control-flow blocks such as loops and if conditions).

Stream API

- Lets you write code:
  - **Declarative** - More concise and readable
  - **Composable** – Greater flexibility
  - **Parallelizable –** Better performance

What exactly the Stream is then?

- A sequence of elements from a source that supports data processing operations.
  - ***Sequence of elements* -** Like a collection, a stream provides an interface to a sequenced set of values of a specific element type.
    - Collections are about data;
    - Streams are about computations;
  - ***Source* -** Streams consume from a data-providing source such as
    - Collections
    - Arrays
    - I/O resources.
  - ***Data Processing Operations* -** Streams support database-like operations and common operations from functional programming languages to manipulate data, such as
    - Filter
    - Map
    - Reduce
    - Find
    - Match
    - Sort
    - and so on.

Two important characteristics

- **Pipelining**— Many stream operations return a stream themselves, allowing operations to be chained and form a larger pipeline.
- A pipeline of operations can be viewed as a database-like query on the data source.
- **Internal iteration**— In contrast to collections, which are iterated explicitly using an iterator, stream operations do the iteration behind the scenes for you.
- **Traversable only once –** Similar to iterators, a stream can only be traversed only once.
  - After the first iteration stream is supposed to be consumed.
  - You can start a new stream on the data source to iterate again.

Example

```java
public static void main(String[] args) {
	List<String> title = Arrays.asList("Sky","is","blue");
	Stream<String> a = title.stream();
	a.forEach(System.out::println);
	a.forEach(System.out::println); // will throw an exception
}
```

Stream operations

```
List<String> names = menu.stream()
                         .filter(d -> d.getCalories() > 300)
                         .map(Dish::getName)
                         .limit(3)
                         .collect(toList());
```

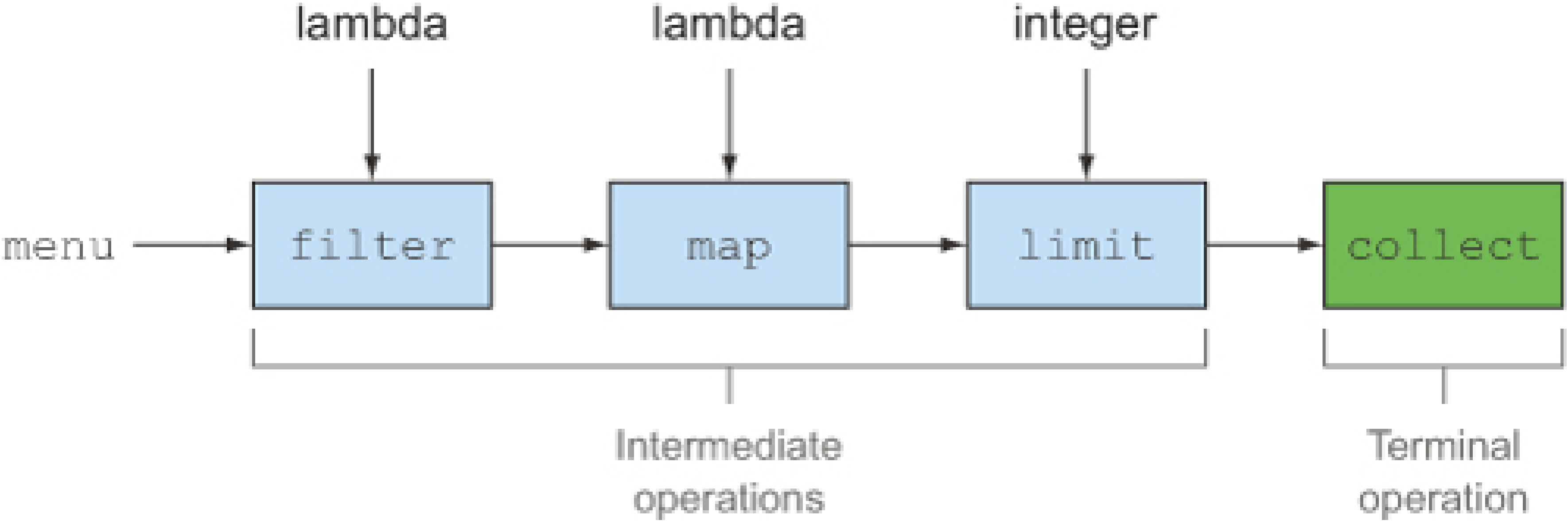
Get a stream from the list of dishes.

Intermediate operation.

Intermediate operation.

Intermediate operation.

Converts the `Stream` into a `List`.



Stream Operations

- Intermediate operations:
  - Intermediate operations such as filter or sorted return another stream as the return type.
  - Intermediate operations don’t perform any processing until a terminal operation is invoked on the stream pipeline—they’re lazy.
- Terminal operations:
  - Terminal operations produce a result from a stream pipeline.
  - A result is any non-stream value such as a List, an Integer, or even void.

Intermediate Stream Operations

Operation	Type	Return type	Argument of the operation	Function descrip-tor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
map	Intermediate	Stream<R>	Function<T, R>	T -> R
limit	Intermediate	Stream<T>		
sorted	Intermediate	Stream<T>	Comparator<T>	(T, T) -> int
distinct	Intermediate	Stream<T>		

Terminal Stream Operations

Operation	Type	Purpose
forEach	Terminal	Consumes each element from a stream and applies a lambda to each of them. The operation returns void.
count	Terminal	Returns the number of elements in a stream. The operation returns a long.
collect	Terminal	Reduces the stream to create a collection such as a List, a Map, or even an Integer. [redacted]