Advanced Programming Methods

Lecture 5 - Functional Programming

Overview

1. Anonymous inner classes in Java
2. Lambda expressions in Java 8
3. Processing Data with Java 8 Streams

Note: Lecture notes are based on Oracle tutorials.

Anonymous Inner classes

- provide a way to implement classes that may occur only once in an application.

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

Functional Interfaces

- are interfaces with only one method
- Using functional interfaces with anonymous inner classes are a common pattern in Java

```
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
```

Lambda Expressions

- are Java's first step into functional programming
- can be created without belonging to any class
- can be passed around as if they were objects and executed on demand.

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

() -> 42

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

testButton.addActionListener(e -> System.out.println("Click Detected by
    Lambda Listner"));
```

Lambda Expressions

- Lambda function body

```
(oldState, newState) -> System.out.println("State changed")


(oldState, newState) -> {
    System.out.println("Old state: " + oldState);
    System.out.println("New state: " + newState);
  }
```

- Returning a value

```
(param) -> {System.out.println("param: " + param); return "return value";}


 (a1, a2) -> { return a1 > a2; }


 (a1, a2) -> a1 > a2;
```

Lambdas as Objects

- A Java lambda expression is essentially an object.
- You can assign a lambda expression to a variable and pass it around, like you do with any other object.

```
public interface MyComparator {

    public boolean compare(int a1, int a2);

}


MyComparator myComparator = (a1, a2) -> return a1 > a2;

boolean result = myComparator.compare(2, 5);
```

Runnable Lambda

```
// Anonymous Runnable
Runnable r1 = new Runnable(){
@Override
public void run(){ System.out.println("Hello world one!"); } };


// Lambda Runnable
 Runnable r2 = () -> System.out.println("Hello world two!");


// Run them!
 r1.run();
 r2.run();
```

Comparator Lambda

```
List<Person> personList = Person.createShortList();

  // Sort with Inner Class

  Collections.sort(personList, new Comparator<Person>(){

    public int compare(Person p1, Person p2){

      return p1.getSurName().compareTo(p2.getSurName());

    }});


  // Use Lambda instead

 Collections.sort(personList, (Person p1, Person p2) →
  p1.getSurName().compareTo(p2.getSurName()));

 Collections.sort(personList, (p1,  p2) ->
  p2.getSurName().compareTo(p1.getSurName()));
```

Lambda Expressions

- can improve your code
- provide a means to better support the Don't Repeat Yourself (DRY) principle
- make your code simpler and more readable.
- Motivational example: Given a list of people, various criteria are used to send messages to matching persons:
  - Drivers(persons over the age of 16) get phone calls
  - Draftees(male persons between the ages of 18 and 25) get emails
  - Pilots(persons between the ages of 23 and 65) get mails

First Attempt

```
public class RoboContactMethods {
  public void callDrivers(List<Person> pl){
     for(Person p:pl){
         if (p.getAge() >= 16){ roboCall(p);}
      } }
 public void emailDraftees(List<Person> pl){
     for(Person p:pl){
        if (p.getAge() >= 18 && p.getAge() <= 25 && p.getGender() == Gender.MALE){
          roboEmail(p);
       } }}
 public void mailPilots(List<Person> pl){
      for(Person p:pl){
        if (p.getAge() >= 23 && p.getAge() <= 65){        roboMail(p);  }
     } }
…...}
```

First Attempt

- The DRY principle is not followed.
    - Each method repeats a looping mechanism.
    - The selection criteria must be rewritten for each method
- A large number of methods are required to implement each use case.
- The code is inflexible. If the search criteria changed, it would require a number of code changes for an update. Thus, the code is not very maintainable.

Second Attempt

```
public class RoboContactMethods2 {
   public void callDrivers(List<Person> pl){
      for(Person p:pl){
         if (isDriver(p)){  roboCall(p);}}}
  public void emailDraftees(List<Person> pl){
       for(Person p:pl){
        if (isDraftee(p)){       roboEmail(p);}}}
 public void mailPilots(List<Person> pl){
     for(Person p:pl){
       if (isPilot(p)){ roboMail(p);}} }
  public boolean isDriver(Person p){ return p.getAge() >= 16; }
   public boolean isDraftee(Person p){
       return p.getAge() >= 18 && p.getAge() <= 25 && p.getGender() == Gender.MALE; }
   public boolean isPilot(Person p){  return p.getAge() >= 23 && p.getAge() <= 65; }
```

Third Attempt

- Using a functional interface and anonymous inner classes

```
public interface Predicate<T> {
   public boolean test(T t);
}


public void phoneContacts(List<Person> pl, Predicate<Person> aTest){
    for(Person p:pl){
       if (aTest.test(p)){    roboCall(p); }
} }


robo.phoneContacts(pl,  new Predicate<Person>(){
           @Override
            public boolean test(Person p){
               return p.getAge() >=16;  } }  );
```

Fourth Attempt

- Using lambda expressions

```
public void phoneContacts(List<Person> pl, Predicate<Person> pred){
   for(Person p:pl){
      if (pred.test(p)){    roboCall(p); }
} }
```

```
Predicate<Person> allDrivers = p -> p.getAge() >= 16;
 Predicate<Person> allDraftees = p -> p.getAge() >= 18 && p.getAge() <= 25 &&
     p.getGender() == Gender.MALE;
 Predicate<Person> allPilots = p -> p.getAge() >= 23 && p.getAge() <= 65;
```

```
robo.phoneContacts(pl, allDrivers);
```

java.util.function

- standard interfaces are designed as a starter set for developers:
  - Predicate: A property of the object passed as argument
  - Consumer: An action to be performed with the object passed as argument
  - Function: Transform a T to a R
  - Supplier: Provide an instance of a T (such as a factory)
  - UnaryOperator: A unary operator from T -> T
  - BinaryOperator: A binary operator from (T, T) -> T

Function Interface

- It has only one method apply with the following signature:

  public R apply(T t)

- Example for class Person:

```
public String printCustom(Function <Person, String> f){

    return f.apply(this);}
```

```
Function<Person, String> westernStyle = p -> {return "\nName: " +
    p.getGivenName() + " " + p.getSurName() + "\n"};

Function<Person, String> easternStyle =  p -> "\nName: " + p.getSurName() + " " +
    p.getGivenName() + "\n"};

person.printCustom(westernStyle);

person.printCustom(easternStyle);

person.printCustom(p -> "Name: " + p.getGivenName() + " EMail: " + p.getEmail());
```

Java 8 Streams

- is a new addition to the Java Collections API, which brings a new way to process collections of objects.
- declarative way
- Stream: a sequence of elements from a source that supports aggregate operations.

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");
myList.stream()
 .filter(s -> s.startsWith("c"))
 .map(String::toUpperCase)
 .sorted()
 .forEach(System.out::println);
```

- Output:

C1

C2

Java 8 Streams

- Sequence of elements: A stream provides an interface to a sequenced set of values of a specific element type. However, streams don't actually store elements; they are computed on demand.

- Source: Streams consume from a data-providing source such as collections, arrays, or I/O resources.

- Aggregate operations: Streams support SQL-like operations and common operations from functional programing languages, such as filter, map, reduce, find, match, sorted, and so on.

Streams vs Collections

- collections are about data
- **streams are about computations**.
- A collection is an in-memory data structure, which holds all the values that the data structure currently has—every element in the collection has to be computed before it can be added to the collection.
- In contrast, a stream is a conceptually fixed data structure in which elements are computed on demand.

Streams vs Collections

Two fundamental characteristics that make stream operations very different from collection operations:

- Pipelining: Many stream operations return a stream themselves. This allows operations to be chained to form a larger pipeline. This enables certain optimizations, such as laziness and short-circuiting
- Internal iteration: In contrast to collections, which are iterated explicitly (external iteration), stream operations do the iteration behind the scenes for you.

Obtaining a Stream From a Collection

```
List<String> items = new ArrayList<String>();

items.add("one");
items.add("two");
items.add("three");

Stream<String> stream = items.stream();
```

- is similar to how you obtain an Iterator by calling the items.iterator() method, but a Stream is different than an Iterator.

Stream Processing Phases

1. **Configuration**-- intermediate operations:
   - filters, mappings
   - can be connected together to form a pipeline
   - return a stream
   - Are lazy: do not perform any processing
2. **Processing**—terminal operations:
   - operations that close a stream pipeline
   - produce a result from a pipeline such as a List, an Integer, or even void (any non-Stream type).

Filtering

- **stream.filter(item -> item.startsWith("o"));**

- filter(Predicate): Takes a predicate (java.util.function.Predicate) as an argument and returns a stream including all elements that match the given predicate
- distinct: Returns a stream with unique elements (according to the implementation of equals for a stream element)
- limit(n): Returns a stream that is no longer than the given size n
- skip(n): Returns a stream with the first n number of elements discarded

Filtering

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);

List<Integer> twoEvenSquares =

    numbers.stream()

           .filter(n -> {System.out.println("filtering " + n); return n % 2 == 0;})

           .map(n -> { System.out.println("mapping " + n); return n * n;})

           .limit(2)

           .collect(toList());
```

filtering 1

filtering 2

mapping 2

filtering 3

filtering 4

mapping 4

- limit(2) uses short-circuiting; we need to process only part of the stream, not all of it, to return a result.

Mapping

- Streams support the method map, which takes a function (java.util.function.Function) as an argument to project the elements of a stream into another form. The function is applied to each element, “mapping” it into a new element.

```
items.stream()
    .map( item -> item.toUpperCase() )
```

- maps all strings in the items collection to their uppercase equivalents.

NOTE: it doesn't actually perform the mapping. It only configures the stream for mapping. Once one of the stream processing methods are invoked, the mapping (and filtering) will be performed.

Mapping

```java
List<String> words = Arrays.asList("Oracle", "Java",
   "Magazine");
 List<Integer> wordLengths =
   words.stream()
       .map(String::length)
       .collect(toList());
```

Stream.collect()

- is an extremely useful terminal operation to transform the elements of the stream into a different kind of result, e.g. a List, Set or Map .

- Collect accepts a Collector which consists of four different operations: a *supplier*, an *accumulator*, a *combiner* and a *finisher*.

- Java 8 supports various builtin collectors via the Collectors class. So for the most common operations you don't have to implement a collector yourself.

Stream.collect()

```
List<Person> filtered =
persons
 .stream()
 .filter(p -> p.name.startsWith("P"))
 .collect(Collectors.toList());


Double averageAge =
persons
 .stream()
 .collect(Collectors.averagingInt(p -> p.age));
```

Stream.collect()

```
String phrase =
persons
 .stream()
 .filter(p -> p.age >= 18)
 .map(p -> p.name)
 .collect(Collectors.joining(" and ", "In Germany ", " are of legal
   age."));
```

- The join collector accepts a delimiter as well as an optional prefix and suffix.

Stream.collect()

- In order to transform the stream elements into a map, we have to specify how both the keys and the values should be mapped.
- the mapped keys must be unique, otherwise an IllegalStateException is thrown.
- You can optionally pass a merge function as an additional parameter to bypass the exception:

```
Map<Integer, String> map = persons
.stream()
.collect(Collectors.toMap(
  p -> p.age,
  p -> p.name,
  (name1, name2) -> name1 + ";" + name2));
```

Stream.min() and Stream.max()

- Are terminal operations
- return an Optional instance which has a get() method on, which you use to obtain the value. In case the stream has no elements the get() method will return null
- take a Comparator as parameter. The Comparator.comparing() method creates a Comparator based on the lambda expression passed to it. In fact, the comparing() method takes a Function which is a functional interface suited for lambda expressions

```
String shortest = items.stream()
        .min(Comparator.comparing(item -> item.length()))
        .get();
```

Stream.min() and Stream.max()

- The Optional<T> class (java.util .Optional) is a container class to represent the existence or absence of a value
- we can choose to apply an operation on the optional object by using the ifPresent method

```
Stream.of("a1", "a2", "a3")
  .map(s -> s.substring(1))
  .mapToInt(Integer::parseInt)
  .max()
  .ifPresent(System.out::println);
```

- Stream.of() creates a stream from a bunch of object references

Stream.count()

- Returns the number of elements in the stream

```
long count = items.stream()
    .filter( item -> item.startsWith("t"))
    .count();
```

Stream.reduce()

- can reduce the elements of a stream to a single value
- takes a BinaryOperator as parameter, which can easily be implemented using a lambda expression.
- Returns an Optional
- The BinaryOperator.apply() method:
  - takes two parameters. The acc which is the accumulated value, and item which is an element from the stream.

```
String reduced2 = items.stream()
        .reduce((acc, item) -> acc + " " + item)
        .get();
```

Stream.reduce()

- There is another reduce() method which takes two parameters: an initial value for the accumulated value, and then a BinaryOperator.

```
String reduced = items.stream()
  .filter( item -> item.startsWith("o"))
  .reduce("", (acc, item) -> acc + " " + item);
```

Stream.reduce()

```
int sum = 0;
for (int x : numbers) {
    sum += x;
}
int sum = numbers.stream().reduce(0, (a, b) -> a + b);

int product = numbers.stream().reduce(1, (a, b) -> a * b);
int max = numbers.stream().reduce(1, Integer::max);
```

Numerical Streams

- IntStream, DoubleStream, and LongStream—that respectively specialize the elements of a stream to be int, double, and long.
- to convert a stream to a specialized version: mapToInt, mapToDouble, and mapToLong.
- to help generate ranges: range and rangeClosed.

```
IntStream oddNumbers =
    IntStream.rangeClosed(10, 30)
        .filter(n -> n % 2 == 1);
```

Building Streams

- InStream<Integer> numbersFromValues = Stream.of(1, 2, 3, 4);

- int[] numbers = {1, 2, 3, 4};

  IntStream numbersFromArray = Arrays.stream(numbers);

- Converting a file into a stream of lines:

  long numberOfLines =

  Files.lines(Paths.get(“yourFile.txt”),Charset.defaultCharset())

  .count();

Infinite Streams

- There are two static methods—Stream.iterate and Stream .generate—that let you create a stream from a function.
- because elements are calculated on demand, these two operations can produce elements “forever.”

```
Stream<Integer> numbers = Stream.iterate(0, n -> n + 10);
```

- The iterate method takes an initial value (here, 0) and a lambda (of type UnaryOperator<T>) to apply successively on each new value produced.

Infinite Streams

- We can turn an infinite stream into a fixed-size stream using the limit operation:

numbers.limit(5).forEach(System.out::println);

// 0, 10, 20, 30, 40.

Finding and Matching

- A common data processing pattern is determining whether some elements match a given property. You can use the **anyMatch, allMatch, and noneMatch** operations to help you do this. They all take a predicate as an argument and return a boolean as the result (they are, therefore, terminal operations)

- Stream interface provides the operations **findFirst and findAny** for retrieving arbitrary elements from a stream. Both findFirst and findAny return an Optional object

Processing Order

```
Stream.of("d2", "a2", "b1", "b3", "c")
 .map(s -> {System.out.println("map: " + s);return s.toUpperCase();})
 .filter(s -> {System.out.println("filter: " + s);return s.startsWith("A");})
 .forEach(s -> System.out.println("forEach: " + s));
```

```
// map: d2
// filter: D2
// map: a2
// filter: A2
// forEach: A2
// map: b1
// filter: B1
// map: b3
// filter: B3
// map: c
// filter: C
```

Processing Order

```
Stream.of("d2", "a2", "b1", "b3", "c")
 .filter(s -> {System.out.println("filter: " + s);return s.startsWith("a");})
 .map(s -> {System.out.println("map: " + s);return s.toUpperCase();})
 .forEach(s -> System.out.println("forEach: " + s));
```

// filter: d2

// filter: a2

// map: a2

// forEach: A2

// filter: b1

// filter: b3

// filter: c

Reusing Streams

- Java 8 streams cannot be reused. As soon as you call any terminal operation the stream is closed

```
Stream<String> stream =
Stream.of("d2", "a2", "b1", "b3", "c")
.filter(s -> s.startsWith("a"));

stream.anyMatch(s -> true); // ok

stream.noneMatch(s -> true);
 // exception since stream has been consumed
```

Reusing Streams

```
Supplier<Stream<String>> streamSupplier =
() -> Stream.of("d2", "a2", "b1", "b3", "c")
.filter(s -> s.startsWith("a"));



streamSupplier.get().anyMatch(s -> true); // ok
streamSupplier.get().noneMatch(s -> true); // ok
```

- *Each call to get() constructs a new stream on which we can call the desired terminal operation.*