Java 8 Streams

Outline

- Lambda Expressions
- Map/Reduce
- Stream Creation
- Stream Operations
- Parallel Streams
- Infinite Streams

Lambda Expressions

- Lambda expressions provide a clear and concise way to represent one method interface using an expression
- Lambda expressions address the bulkiness of anonymous inner classes by converting multiple lines of code into a single statement.

Lambda Expressions

- A lambda expression is composed of three parts.
  - Argument list ()
  - Arrow Token
  - Body

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

() -> 42

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

Lambda Expressions

- Here is how you write a Runnable using lambdas.

```
public class RunnableTest {
  public static void main(String[] args) {

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

    Thread t=new Thread(r2);
    t.start();
  }
}
```

Lambda Expressions

- The following code applies a Comparator by using a couple of lambda expressions

```
public class ComparatorTest {

  public static void main(String[] args) {

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

    System.out.println("=== Sorted Asc SurName ===");
    Collections.sort(personList, (Person p1, Person p2) -> p1.getSurName().compareTo(p2.getSurName()));

    for(Person p:personList)
      p.printName();

    System.out.println("=== Sorted Desc SurName ===");
    Collections.sort(personList, (p1, p2) -> p2.getSurName().compareTo(p1.getSurName()));

    for(Person p:personList)
      p.printName();
  }
}
```

Lambda Expressions

- Given a list of people, various criteria are used to make robo calls (automated phone calls) to matching persons. Our message needs to get out to three different groups
  - Drivers: Persons over the age of 16
  - Draftees: Male persons between the ages of 18 and 25
  - Pilots: Persons between the ages of 23 and 65

Lambda Expressions

- Given a list of people, various criteria are used to make robo calls (automated phone calls) to matching persons. Our message needs to get out to three different groups

```
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);
  }
```

Lambda Expressions

- Lambda expressions allow the easy reuse of any expression.

```
public class RoboCallTest {

    public static void main(String[] args){

      List<Person> pl = Person.createShortList();
      RoboContactLambda robo = new RoboContactLambda();

      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;

      System.out.println("\n=== Calling all Drivers ===");
      robo.phoneContacts(pl, allDrivers);

      System.out.println("\n=== Emailing all Draftees ===");
      robo.emailContacts(pl, allDraftees);

      System.out.println("\n=== Mail all Pilots ===");
      robo.mailContacts(pl, allPilots);
    }
  }
```

java.util.function Package

- Predicate is not the only functional interface provided with Java 8. A number of standard interfaces are designed as a starter set
  - 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 U
  - 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

java.util.function Package

- Here is an example of how to implement a Person printing

```
public class NameTestNew {

  public static void main(String[] args) {

    List<Person> list1 = Person.createShortList();


    Function<Person, String> westernStyle =
            p -> {
            return "\nName: " + p.getGivenName() + " "
                + p.getSurName() + "\n" + "Age: " + p.getAge()
                + "  " + "Gender: " + p.getGender() + "\n"
                + "EMail: " + p.getEmail() + "\n"
                + "Phone: " + p.getPhone() + "\n"
                + "Address: " + p.getAddress();
    };

    Function<Person, String> easternStyle =
        p ->{ return "\nName: " + p.getSurName() + " "
            + p.getGivenName() + "\n" + "Age: " + p.getAge() + "  " +
            "Gender: " + p.getGender() + "\n" +
            "EMail: " + p.getEmail() + "\n" +
            "Phone: " + p.getPhone() + "\n" +
            "Address: " + p.getAddress(); };
```

java.util.function Package

- Lambda expressions allow the easy reuse of any expression.

```
public class NameTestNew {

  public static void main(String[] args) {

    System.out.println("\n===Western List===");
    for (Person person:list1){
        System.out.println(
            person.printCustom(westernStyle)
        );
    }

    System.out.println("\n===Eastern List===");
    for (Person person:list1){
        System.out.println(
            person.printCustom(easternStyle)
        );
    }

  }
}

// Added to Person class
public String printCustom(Function <Person, String> f){
    return f.apply(this);
}
```

MapReduce

- Map/Reduce anaçatısı ile paralel bir biçimde işlemci öbekleri ile iş yapılabilir
- **Map:** Ana düğüm problemi küçük parçalara ayırır ve çalışan düğümlere dağıtır
- **Reduce:** Ana düğüm tüm alt düğümler cevapları alır ve birleştirir.

1. Veriyi parçalara böl (chunk)
2. Tüm chunklar üzerinde paralel olarak **Map** çalıştır
3. Map işlemlerinin sonucunu düzenle ve Reduce'a besle
4. Paralel olarak **Reduce** çalıştır
5. Reduce işlemlerinin sonucunu düzenle

Map-Reduce

- <key, value> çiftleri
- Girdi – çıktı çiftleri farklı tiplerde olabilir
- Key:
  - Writable (serialize), Comparable (sort)
- Value:
  - Writable (serialize)

Kelime Sayma

Map

```
public void map (LongWritable key, Text value,
                 OutputCollector<Text, IntWritable> output,
                 Reporter reporter){

    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);

    while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, new IntWritable(1));
    }
}
```

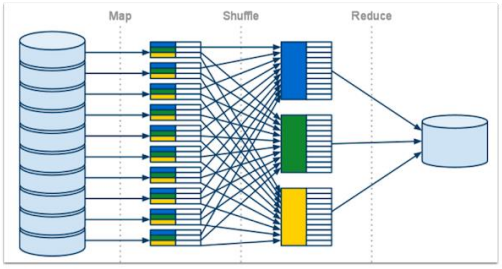
Reduce

```
public void reduce(Text key, Iterator<IntWritable> values,
                   OutputCollector<Text, IntWritable>
output,
                   Reporter reporter) {

    int sum = 0;
    while (values.hasNext()) {
        sum += values.next().get();
    }

    output.collect(key, new IntWritable(sum));
}
```

Main

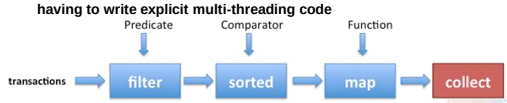
```
JobConf conf = new JobConf(WordCount.class);
conf.setJobName("wordcount");

conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);
conf.setMapperClass(Map.class);

conf.setCombinerClass(Reduce.class);
conf.setReducerClass(Reduce.class);

conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);

JobClient.runJob(conf);
```

MapReduce



Streams

- **The Java 8 contains a new abstraction called Stream that lets you process data in a declarative way. Furthermore, streams can leverage multi-core architectures without you having to write a single line of multithread code.**
  - **Streams have no storage. They carry values from a source through a pipeline of operations.**
  - **They also never modify the underlying data structure.**
  - **Lazy evaluation: Many Stream operations are postponed until it is known how much data is eventually needed**
  - **Parallelizable:If you designate a Stream as parallel, then operations on it will automatically be done concurrently, without having to write explicit multi-threading code**



Stream Creation

- **When you make or transform a Stream, it does not copy the underlying data. Instead, it just builds a pipeline of operations.**
- **Three most common ways to make Stream**
  - **From Lists**

```
List<Student> students= ...;
students.stream().map(...).filter(...).other(...);
```

  - **From arrays**

```
Student[] students= ...;
Stream.of(students).map(...).filter(...).other(...);
```

  - **From individual elements**

```
Student s1 = ...; Student s2 = ...;
Stream.of(s1,s2, ...).map(...).filter(...).other(...);
```

- **Turning Streams into Pre-Java-8 Data Structures**
  - **List**

```
someStream.collect(Collectors.toList())
```

  - **Array**

```
studentStream.toArray(Student[]::new)
```

Stream Creation

- **Caution: Use Objects, not Primitives**
  - **Producing IntStream by accident**

```
int[] nums = { 1, 2, 3, 4 };
Arrays.stream(nums)... // IntStream
Integer[] nums = { 1, 2, 3, 4 };
Arrays.stream(nums)... or Stream.of(nums)... // Stream<Integer>
```

  - **Making 1-item stream by accident**

Wrong

```
int[] nums = { 1, 2, 3, 4 };
Stream.of(nums)... // 1-item Stream containing array
```

Right

```
Integer[] nums = { 1, 2, 3, 4 };
Stream.of(nums)... // 4-item Stream containing Integers
```

Stream Operations

- **Operations that appear to traverse Stream multiple times actually traverse it only once.**
- **Three types of stream methods:**
  - **Intermediate methods: These are methods that produce other Streams. These methods don't get processed until there is some terminal method called.**
  - **Terminal methods: After one of these methods is invoked, the Stream is considered consumed and no more operations can be performed on it.**
  - **Short-circuit methods: These methods cause the earlier intermediate methods to be processed only until the short-circuit method can be evaluated.**

Stream Operations - forEach

- Easy way to loop over Stream elements.
- You supply a lambda to forEach, and that lambda is called on each element of the Stream
  - More precisely, you supply a Consumer to forEach, and each element of the Stream is passed to that Consumer's accept method.
- There is also "peek" method, which is almost exactly the same as forEach, except it returns the original Stream at the end.

```
Stream.of(someArray).forEach(System.out::println);
fieldList.stream().forEach(field -> field.setText(""));
```

- forEach is a "terminal operation", which means that it consumes the elements of the Stream. You cannot apply forEach twice!
- You cannot change values of surrounding local variables
- You cannot break out of the loop early

Stream Operations - forEach

```
employees.stream().forEach(System.out::println);

employees.stream().forEach(e -> e.setSalary(e.getSalary() * 11/10));

employees.stream().forEach(System.out::println);

Consumer<Employee> giveRaise = e -> {
                System.out.printf("%s earned $%,d before raise.%n",
                                e.getFullName(), e.getSalary());

                e.setSalary(e.getSalary() * 11/10);

                System.out.printf("%s will earn $%,d after raise.%n",
                                e.getFullName(), e.getSalary());
                };

employees.stream().forEach(giveRaise);
```

Stream Operations - map

- Produces a new Stream that is the result of applying a Function to each element of original Stream.

```
Double[] nums = { 1.0, 2.0, 3.0, 4.0, 5.0 };
Double[] squares = Stream.of(nums)
                        .map(n -> n * n).toArray(Double[]::new);
nums = Stream.of(squares).map(Math::sqrt).toArray(Double[]::new);

Integer[] ids = { 1, 2, 4, 8 };
List<Employee> matchingEmployees =
Stream.of(ids).map(EmployeeUtils::findById)
        .map(Employee::getFullName).collect(Collectors.toList());
```

Stream Operations - filter

- Produces a new Stream that contain only the elements of the original Stream that pass a given test

```
Integer[] nums = { 1, 2, 3, 4, 5, 6 };
Integer[] evens = Stream.of(nums).filter(n -> n%2 == 0)
                                .filter(n -> n>3)
                                .toArray(Integer[]::new);

Integer[] ids = { 16, 8, 4, 2, 1 };
Stream.of(ids).map(EmployeeUtils::findById)
        .filter(e -> e != null)
        .filter(e -> e.getSalary() > 500000));
```

Stream Operations - findFirst

- Returns an Optional for the first entry in the Stream. Since Streams are often results of filtering, there might not be a first entry, so the Optional could be empty.
  - Optional either stores a T or stores nothing. Useful for methods that may or may not find a value.
- There is also a similar findAny method, which might be faster for parallel Streams.
- Due to lazy evaluation, map or filter know to only find a single match and then stop.

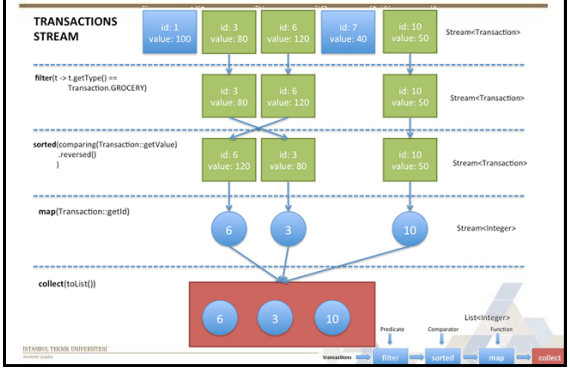
```
Integer[] ids = { 16, 8, 4, ...}; // 10,000 entries
System.out.printf("First Employee with salary over $500K: %s%n",
                Stream.of(ids).map(EmployeeUtils::findEmployee)
                        .filter(e -> e != null)
                        .filter(e -> e.getSalary() > 500000)
                        .findFirst()
                        .orElse(null));
```

Stream Operations

```
Integer[] ids = { 16, 8, 4, ...}; // 10,000 entries
System.out.printf("First Employee with salary over $500K: %s%n",
                Stream.of(ids).map(EmployeeUtils::findEmployee)
                        .filter(e -> e != null)
                        .filter(e -> e.getSalary() > 500000)
                        .findFirst()
                        .orElse(null));
```

- Apparent behavior
  - findById on all, check all for null, call getSalary on all non-null (& compare to $500K) on all remaining, find first, return it or null
- Actual behavior
  - findById on first, check it for null, if pass, call getSalary, if salary > $500K, return and done. Repeat for second, etc. Return null if you get to the end and never got match.

Stream Operations



Stream Operations - reduce

- You start with a seed (identity) value, combine this value with the first entry of the Stream, combine the result with the second entry of the Stream, and so forth.
  nums.stream().reduce(Double.MIN_VALUE, Double::max)
  nums.stream().reduce(1.0, (n1, n2) -> n1 * n2)
- Concatanating strings

```
List<String> letters = Arrays.asList("a", "b", "c", "d");

letters.stream().reduce("", String::concat);
letters.stream().reduce("", (s1,s2) -> s2+s1);
letters.stream().reduce("",(s1,s2) -> s2.toUpperCase() +
s1.toUpperCase());
```

Stream Operations - reduce

- Employee Operations

```
Employee minSalary = new Employee("None", "None", -1, -1);
BinaryOperator<Employee> moreSalary = (e1, e2) -> {
                    return(e1.getSalary() >= e2.getSalary() ? e1 : e2);
                  };

Employee maxsalary= employees.stream().reduce(minSalary , moreSalary
);
System.out.printf("Max salary employee is %s.%n", maxsalary);
```

- Finding Biggest Number

```
List<Double> nums = Arrays.asList(1.2, -2.3, 4.5, -5.6);
double maxNum = nums.stream().reduce(Double.MIN_VALUE,
Double::max);

double[] nums = {1.2, -2.3, 4.5, -5.6};
double maxNum =
DoubleStream.of(nums).max().orElse(Double.MIN_VALUE);
```

Stream Operations - reduce

- Summing Numbers

```
List<Integer> nums2 = Arrays.asList(1, 2, 3, 4);
int sum1 = nums2.stream().reduce(0, Integer::sum);

int sum2 = nums2.stream().reduce(Integer::sum).orElse(0);

int[] nums3 = { 1, 2, 3, 4 };
int sum3 = Arrays.stream(nums3).sum();
// Or IntStream.of(nums3).sum()
```

- Combining map and reduce

```
int sum4 = nums2.stream().map(EmployeeUtils::findEmployee)
                         .map(Employee::getSalary)
                         .reduce(0, Integer::sum);

int sum4 = nums2.stream().map(EmployeeUtils::findEmployee)
                         .map(Employee::getSalary)
                         .reduce(0, Integer::sum);

int sum5 = nums2.stream().map(EmployeeUtils::findEmployee)
                         .mapToInt(Employee::getSalary)
                         .sum();
```

Stream Operations - limit

- limit(n) returns a Stream of the first n elements, skip(n) returns a Stream starting with element n

```
List<String> emps = employees.stream()
                         .map(Person::getFirstName)
                         .limit(8)
                         .skip(2)
                         .collect(Collectors.toList());
```

- getFirstName is called only 6 times, even if Stream has 10,000 elements

Stream Operations - sorting

- Doing limit or skip after sorting does not short-circuit in the same manner as in the previous
- If the Stream elements implement Comparable, you may omit the lambda and just use someStream.sorted().

```
List<Integer> ids = Arrays.asList(9, 11, 10, 8);
List<Employee> emps = ids.stream()
                         .map(EmployeeUtils::findEmployee)
                         .sorted((e1, e2) ->

e1.getLastName().compareTo(e2.getLastName()))
                         .collect(Collectors.toList());

List<Employee> emps =
employees.sorted(Person::firstNameComparer)
                         .limit(2)
                         .collect(Collectors.toList());
```

Stream Operations - min max

- min and max both return an Optional
- Unlike with sorted, you must provide a lambda, regardless of whether or not the Stream elements implement Comparable
- min and max are O(n), sorted is O(n log n)

```
Employee e = ids.stream().map(EmployeeUtils::findEmployee)
                         .min((e1, e2) ->

e1.getLastName().compareTo(e2.getLastName()))
                         .get();

Employee e = ids.stream().map(EmployeeUtils::findEmployee)
                         .max((e1, e2) -> e1.getSalary()-e2.getSalary())
                         .get();

List<Integer> ids = Arrays.asList(9, 10, 9, 10, 9, 10);
List<Employee> emps = ids.stream().map(EmployeeUtils::findEmployee)
                         .distinct()
                         .collect(Collectors.toList());
```

Stream Operations - matching

- allMatch, anyMatch, and noneMatch take a Predicate and return a boolean. They stop processing once an answer can be determined.
- count simply returns the number of elements

```
boolean isLowSalary = employees.stream().noneMatch(e ->
e.getSalary() < 200000);

Predicate<Employee> highSalary = e -> e.getSalary() > 7000000;

boolean isOneHighSalary = employees.stream().anyMatch(highSalary);

boolean isAllHighSalary = employees.stream().allMatch(highSalary);

long highSalaryCount = employees.stream().filter(highSalary).count();
```

Stream Operations - collect

- **Combined with methods in the Collectors class, you can build many data types out of Streams**
  - **List**
    anyStream.collect(toList())
  - **String**
    stringStream.collect(joining(delimiter)).toString()
  - **Set**
    anyStream.collect(toSet())
  - **Collection**
    anyStream.collect(toCollection(CollectionType::new))
  - **Map**
    strm.collect(partioningBy(...)), strm.collect(groupingBy(...))

Stream Operations - collect

- Strings

```
List<Integer> ids = Arrays.asList(2, 4, 6, 8);
String lastNames = ids.stream().map(EmployeeUtil::findEmployee)
                              .map(Employee::getLastName)
                              .collect(Collectors.joining(", "))
                              .toString();
```

- Sets

```
Set<String> firstNames =
employees.stream().map(Employee::getFirstName)
                                  .collect(Collectors.toSet());

Stream.of("Larry", "Harry", "Peter", "Deiter", "Eric", "Barack")
    .forEach(s -> System.out.printf("%s is an Employee? %s.%n", s,
                                  firstNames.contains(s) ? "Yes" : "No"));
```

Stream Operations - collect

- Collections

```
TreeSet<String> firstNames =
employees.stream().map(Employee::getFirstName)
                  .collect(Collectors.toCollection(TreeSet::new));
```

- Maps:
  - partitioningBy: You provide a Predicate. It builds a Map where true maps to a List of entries that passed the Predicate, and false maps to a List that failed the Predicate.

```
Map<Boolean,List<Employee>> highSalaryTable =
employees.stream().collect(partitioningBy(e -> e.getSalary() >
1000000));
System.out.printf("High salary employees: %s.%n",
highSalaryTable.get(true));
System.out.printf("The rest: %s.%n", highSalaryTable.get(false));
```

Stream Operations - collect

- Maps:
  - groupingBy: You provide a Function. It builds a Map where each output value of the Function maps to a List of entries that gave that value.

```
Map<String,List<Employee>> officeTable = employees.stream()
                              .collect(Collectors.groupingBy(Emp::getOffice));

System.out.printf("Emps in LA: %s.%n",officeTable.get("Mountain
View"));

System.out.printf("Emps in NY: %s.%n",officeTable.get("New York"));

System.out.printf("Emps in Zurich: %s.%n",officeTable.get("Zurich"));
```

Parallel Streams

- By designating that a Stream be parallel, the operations are automatically done in parallel. No explicit multi-threading code is required.

someStream.forEach(someThreadSafeOp)
someStream.parallel().forEach(someThreadSafeOp)

- Combining operation should be associative to be parallelized
  - **(a op b) op c == a op (b op c)**
- No side effects on global data are performed
- All parallel streams use common fork-join thread pool !!!
- Lambda expressions in stream operations should not interfere. Interference occurs when the source of a stream is modified while a pipeline processes the stream.
- Avoid using stateful lambda expressions as parameters in stream operations. A stateful lambda expression is one whose result depends on any state that might change during the execution of a pipeline.

Parallel Streams

```
double average = roster.parallelStream()
                       .filter(p -> p.getGender() == Person.Sex.MALE)
                       .mapToInt(Person::getAge).average().getAsDouble();

Map<Person.Sex, List<Person>> byGender =
   roster
      .stream()
      .collect(Collectors.groupingBy(Person::getGender));

ConcurrentMap<Person.Sex, List<Person>> byGender =
   roster
      .parallelStream()
      .collect(Collectors.groupingByConcurrent(Person::getGender));
```

- This is called a concurrent reduction. The Java runtime performs a concurrent reduction if all of the the following are true for a particular pipeline that contains the collect operation:
  - The stream is parallel.
  - The collector, has the characteristic Collector.Characteristics.CONCURRENT.
  - Either the stream is unordered, or the collector has the characteristic Collector.Characteristics.UNORDERED.

Infinite Streams - suppliers

- You supply a function (Supplier) to Stream.generate. Whenever the system needs stream elements, it invokes the function to get them.
  - You must limit the Stream size usually with limit.
  - The function can maintain state so that new values are based on any or all of the previous values

```
Supplier<Double> random = Math::random;
System.out.println("2 Random numbers:");
Stream.generate(random).limit(2).forEach(System.out::println);
System.out.println("4 Random numbers:");
Stream.generate(random).limit(4).forEach(System.out::println);

List<Employee> emps = Stream.generate(() -> randomEmployee())
                            .limit(10).collect(Collectors.toList());
```

Infinite Streams - iterators

- You specify a seed value and a UnaryOperator f. The seed becomes the first element of the Stream, f(seed) becomes the second element, f(second) becomes third element, etc.
  - You must limit the Stream size usually with limit.
  - The function can maintain state so that new values are based on any or all of the previous values

```
Stream.iterate("Big News!!", msg -> msg + "!!!!!!!!!!")
    .limit(14).forEach(System.out::println);

Stream.iterate(Primes.findPrime(numDigits),Primes::nextPrime).limit(1000)
```

Thanks

- Slides have been adoted from CoreServlets Tutorials and Oracle Java By Example Tutorials
- http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html
- http://www.coreservlets.com/java-8-tutorial/#streams-1