Object-oriented programming

Lecture №7

lambda functions, Stream API

PhD, Alexander Vlasov

Question?

- Collections
- Iterators
- Java’s containers
- Algorithms for collection

Лямбда-выражения

```
class LengthStringComparator implements Comparator<String> {
    public int compare(String firstStr, String secondStr) {
        return Integer.compare(firstStr.length(),secondStr.length());
    }
}
Arrays.sort(strings, new LengthStringComparator ());
```

Лямбда-выражения

- (String firstStr, String secondStr) ->
  Integer.compare(firstStr.length(),secondStr.length());
- Comparator<String> comp = (firstStr, secondStr) ->
  Integer.compare(firstStr.length(),secondStr.length());
- (int x) -> { if (x <= 1) return -1; }
- Arrays.sort(strs, (firstStr, secondStr) ->
  Integer.compare(firstStr.length(), secondStr.length()));
- Arrays.sort(strs, String::compareToIgnoreCase)

Лямбда-выражения и Java

- Является блоком кода с параметрами
- Выполнение блока кода в более поздний момент времени
- Могут быть преобразованы в функциональные интерфейсы.
- Имеют доступ к final переменным из охватывающей области видимости.
- Ссылки на реализованные методы и конструкторы

Примеры Лямбда-выражения

```
int[]::new  //  x -> new int[x]
```

```
public static void main(String[] args) {
   int n=70;
   int m=30;
   Operation op = ()->{
      //n=100; - error
      return m+n;
   };
   // n=100;  - error
   System.out.println(op.calculate()); // 100
}
```

```
Operationable operation = (int x, int y)-> {
   if(y==0)
      return 0;
   else
      return x/y;
};
System.out.println(operation.calculate(20, 10)); //2
System.out.println(operation.calculate(20, 0));   //0
```

```
Operationable operation1 = (int x, int y)-> x + y;
Operationable operation2 = (int x, int y)-> x - y;
Operationable operation3 = (int x, int y)-> x * y;

System.out.println(operation1.calculate(20, 10)); //30
System.out.println(operation2.calculate(20, 10)); //10
System.out.println(operation3.calculate(20, 10)); //200
```

```
public class LambdaApp {
   static int x = 10;
   static int y = 20;
   public static void main(String[] args) {
      Operation op = ()->{
         x=30;
         return x+y;
      };
      System.out.println(op.calculate()); // 50
      System.out.println(x); // 30
   }
}
interface Operation{
   int calculate();
}
```

Типичные задачи

- Инициализация коллекции
- Преобразование в другой тип коллекции
- Выборка нужных элементов
- ….

- Stream API - classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections.
- java.util.stream

Stream API

```
List<String> strs = ...;
Stream<Button> stream = strs.stream().map(Button::new);  // Button(String)
List<Button> buttons = stream.collect(Collectors.toList());
```

```
Object[] buttons = stream.toArray(); //How is array created?
Button[] buttons = stream.toArray(Button[]::new);  //Create Array
```

Modifier and Type	Method and Description
boolean	allMatch(Predicate<? super T> predicate)Returns whether all elements of this stream match the provided predicate.
boolean	anyMatch(Predicate<? super T> predicate)Returns whether any elements of this stream match the provided predicate.
<R,A> R	collect(Collector<? super T,A,R> collector)Performs a **mutable reduction** operation on the elements of this stream using a Collector.
<R> R	collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)Performs a **mutable reduction** operation on the elements of this stream.
static <T> Stream<T>	concat(Stream<? extends T> a, Stream<? extends T> b)Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream.
long	count()Returns the count of elements in this stream.
Stream<T>	distinct()Returns a stream consisting of the distinct elements (according to **Object.equals(Object)**) of this stream.
static <T> Stream<T>	empty()Returns an empty sequential Stream.
Stream<T>	filter(Predicate<? super T> predicate)Returns a stream consisting of the elements of this stream that match the given predicate.
Optional<T>	findAny()Returns an **Optional** describing some element of the stream, or an empty Optional if the stream is empty.
Optional<T>	findFirst()Returns an **Optional** describing the first element of this stream, or an empty Optional if the stream is empty.
void	forEach(Consumer<? super T> action)Performs an action for each element of this stream.
void	forEachOrdered(Consumer<? super T> action)Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
static <T> Stream<T>	generate(Supplier<T> s)Returns an infinite sequential unordered stream where each element is generated by the provided Supplier.
static <T> Stream<T>	iterate(T seed, UnaryOperator<T> f)Returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc.
Stream<T>	limit(long maxSize)Returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.
<R> Stream<R>	map(Function<? super T,? extends R> mapper)Returns a stream consisting of the results of applying the given function to the elements of this stream.
Optional<T>	max(Comparator<? super T> comparator)Returns the maximum element of this stream according to the provided Comparator.
Optional<T>	min(Comparator<? super T> comparator)Returns the minimum element of this stream according to the provided Comparator.
boolean	noneMatch(Predicate<? super T> predicate)Returns whether no elements of this stream match the provided predicate.
static <T> Stream<T>	of(T... values)Returns a sequential ordered stream whose elements are the specified values.
static <T> Stream<T>	of(T t)Returns a sequential Stream containing a single element.
Stream<T>	peek(Consumer<? super T> action)Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.
Optional<T>	reduce(BinaryOperator<T> accumulator)Performs a **reduction** on the elements of this stream, using an **associative** accumulation function, and returns an Optional describing the reduced value, if any.
T	reduce(T identity, BinaryOperator<T> accumulator)Performs a **reduction** on the elements of this stream, using the provided identity value and an **associative** accumulation function, and returns the reduced value.
<U> U	reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)Performs a **reduction** on the elements of this stream, using the provided identity, accumulation and combining functions.
Stream<T>	skip(long n)Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream.
Stream<T>	sorted()Returns a stream consisting of the elements of this stream, sorted according to natural order.
Stream<T>	sorted(Comparator<? super T> comparator)Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.
Object[]	toArray()Returns an array containing the elements of this stream.
<A> A[]	toArray(IntFunction<A[]> generator)Returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.

Нахождение максимального и минимального значений

```
ArrayList<Integer> testValues = new ArrayList();
testValues.add(0,15);
testValues.add(1,1);
testValues.add(2,2);
testValues.add(3,100);
testValues.add(4,50);

Optional<Integer> maxValue = testValues.stream().max(Integer::compareTo);
System.out.println("MaxValue="+maxValue);
Optional<Integer> minValue = testValues.stream().min(Integer::compareTo);
System.out.println("MinValue="+minValue);
```

Нахождение максимального значения исключая null

```
ArrayList<Integer> testValuesNull = new ArrayList();
testValuesNull.add(0,null);
testValuesNull.add(1,1);
testValuesNull.add(2,2);
testValuesNull.add(3,70);
testValuesNull.add(4,50);

Optional<Integer> maxValueNotNull =  testValuesNull.stream().filter((p) -> p != null).max(Integer::compareTo);
System.out.println("maxValueNotNull="+maxValueNotNull);
```

Найти количество записей с именем Ivan

```
Collection<SportsCamp> sport = Arrays.asList(
        new SportsCamp("Ivan", 5),
        new SportsCamp( null, 15),
        new SportsCamp("Petr", 7),
        new SportsCamp("Ira", 10)
);
```

```
Stream streamSport = Stream.of(
        new SportsCamp("Ivan", 5),
        new SportsCamp( null, 15),
        new SportsCamp("Petr", 7),
        new SportsCamp("Ira", 10)
)
```

```
long countName =  sport.stream().filter((p) -> p.getName() != null && p.getName().equals("Ivan")).count();
System.out.println("countName="+countName);

long countNameParallel = sport.parallelStream().filter((p) -> p.getName() != null && p.getName().equals("Ivan")).count();
System.out.println("countNameParallel=" + countNameParallel);
```

Collectors

```
// Accumulate names into a List
List<String> list = people.stream().map(Person::getName).collect(Collectors.toList());
// Accumulate names into a TreeSet
Set<String> set = people.stream().map(Person::getName).collect(Collectors.toCollection(TreeSet::new));
// Convert elements to strings and concatenate them, separated by commas
String joined = things.stream().map(Object::toString).collect(Collectors.joining(", "));
// Compute sum of salaries of employee
int total = employees.stream().collect(Collectors.summingInt(Employee::getSalary)));
// Group employees by department
Map<Department, List<Employee>> byDept
    = employees.stream().collect(Collectors.groupingBy(Employee::getDepartment));
// Compute sum of salaries by department
Map<Department, Integer> totalByDept
    = employees.stream().collect(Collectors.groupingBy(Employee::getDepartment,
Collectors.summingInt(Employee::getSalary)));
// Partition students into passing and failing
Map<Boolean, List<Student>> passingFailing =students.stream() .collect(Collectors.partitioningBy(s ->
s.getGrade() >= PASS_THRESHOLD));
```

MyCollector

```
StringBuilder b = new StringBuilder(); // метод_инициализации_аккумулятора
for(String s: strings) {
   b.append(s).append(" , "); // метод_обработки_каждого_элемента,
}
String joinBuilderOld = b.toString(); // метод_последней_обработки_аккумулятора
```

```
Collector<Тип_источника, Тип_аккумулятора, Тип_результата> collector =  Collector.of(
        метод_инициализации_аккумулятора,
        метод_обработки_каждого_элемента,
        метод_соединения_двух_аккумуляторов,
        [метод_последней_обработки_аккумулятора]
    );
```

```
String joinBuilder = strings.stream().collect(
  Collector.of(
        StringBuilder::new, // метод_инициализации_аккумулятора
        (b ,s) -> b.append(s).append(" , "), // метод_обработки_каждого_элемента,
        (b1, b2) -> b1.append(b2).append(" , "), // метод_соединения_двух_аккумуляторов
        StringBuilder::toString // метод_последней_обработки_аккумулятора
    )
);
```

Свой Collectors.toList()

```
// Напишем свой аналог toList
    Collector<String, List<String>, List<String>> toList = Collector.of(
        ArrayList::new, // метод инициализации аккумулятора
        List::add, // метод обработки каждого элемента
        (l1, l2) -> { l1.addAll(l2); return l1; } // метод соединения двух аккумуляторов при параллельном выполнении
    );
// Используем его для получение списка строк без дубликатов из стрима
    List<String> distinct1 = strings.stream().distinct().collect(toList);
```

Reduce

Метод reduce позволяет выполнять агрегатные функции на всей коллекцией (такие как сумма, нахождение минимального или максимального значение и т.п.), он возвращает одно значение для стрима, функция получает два аргумента — значение полученное на прошлых шагах и текущее значение.

Arrays.asList(1, 2, 3, 4, 2)

```
collection.stream().reduce((s1, s2) -> s1 + s2).orElse(0)
collection.stream().reduce(Integer::max).orElse(-1)
collection.stream().filter(o -> o % 2 != 0).reduce((s1, s2) -> s1 + s2).orElse(0)
```

Some examples

- Stream<Integer> numbers = Stream.iterate(0, n -> n + 10);
- numbers.limit(5).forEach(System.out::println); // 0, 10, 20, 30, 40
- IntStream oddNumbers = IntStream.rangeClosed(10, 30) .filter(n -> n % 2 == 1);
- int product = numbers.stream().reduce(1, (a, b) -> a * b);
- List<Integer> wordLengths = words.stream().map(String::length).collect(toList());
- Map<Currency, List<Transaction>> transactionsByCurrencies =transactions.stream().collect(groupingBy(Transaction::getCurrency));

Duck typing

If it walks like a duck and it quacks like a duck, then it must be a duck

```
class Duck:
    def fly(self):
        print("Duck flying")

class Airplane:
    def fly(self):
        print("Airplane flying")

class Whale:
    def swim(self):
        print("Whale swimming")

for animal in Duck(), Airplane(), Whale():
    animal.fly()
```

```
Duck flying
Airplane flying
AttributeError: 'Whale' object has no attribute 'fly'
```

- Теперь вы можете добавить методы по умолчанию и статические методы к интерфейсам, которые обеспечивают конкретные реализации.
- Вы должны разрешать любые конфликты между методами по умолчанию из нескольких интерфейсов.