Lambda in Java

Сирота Елена

Java 8

- Выпуск планируется летом 2013
- В Java 8 грядут наиболее значительные изменения за все время существования Java
  - Поддержка lambda (closures)
  - Расширение интерфейсов (default methods)
  - Эволюция Collections (bulk operations)
  - Поддержка параллельности выполнения на уровне библиотек

Как это «пощупать»

- OpenJDK 8

Времена меняются

- В 1995 (когда была создана Java) наиболее популярные языки не поддерживали lambda
- Сегодня:
  - C++ поддерживает lambda
  - C# поддерживает lambda
  - Любой новый язык поддерживает lambda

Лямбда-исчисление

- Математическое понятие выражения (функции)
  - square : REAL → REAL
  - "+": [REAL x REAL] → REAL

- Лямбда-исчисление вводит понятие функции следующим образом:
  - square λ x: REAL | x * x
    - Задается **тело функции**
    - Как аргументы (x) могут быть использованы другие функции

- До появления Java 8 и lambda для подобных задач в Java использовались анонимные классы

Пример анонимного класса

```
interface ActionListener {
    void actionPerformed(ActionEvent a);
}

public class Controller {

    public init() {
        …
        button.addActionListener( new ActionListener() {
            public void actionPerfored(ActionEvent e) {
                // do something.
            }
        });
}
```

Пример анонимного класса

```
interface Runnable { void run(); }



Thread t = new Thread (new Runnable() {
    void run () {
        System.out.println("hello");
    }
});
t.start();
```

Что такое lambda-выражение

- Labmda-выражение – это анонимный метод
  - Имеет список аргументов и тело
- Примеры

```
(Object o) -> o.toString()

s -> s.length()

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

() -> 42

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

Как «вызвать» lambda-выражение

- Ожидается что-то похожее на

```
{ int x => x + 1 }.invoke(10)


int sum = { int x, int y => x + y }.invoke(3, 4);
```

- **Но в Java 8 не так**

Как «вызвать» lambda-выражение

- С помощью функционального интерфейса
- Функциональный интерфейс – это интерфейс, который содержит только один метод

- Вызвать lambda-выражение означает инстанцировать функциональный интерфейс

- Пример функционального интерфейса:

```
interface Runnable { void run(); }
```

- Пример вызова лямбда-выражения

```
Runnable r = () -> { System.out.println("hello"); };
Thread t = new Thread (r);
t.start();
```

Примеры функциональных интерфейсов

- Функциональный интерфейс

```
interface ActionListener {
    void onEvent(Event e);
}
```

- Задаем Lambda-выражение:

```
ActionListener listenr = e -> System.out.println(e.getWhen());
```

- Передаем Lambda-выражение для вызова:

```
button.addActionListener(listener);
```

Примеры функциональных интерфейсов

- Пример функционального интерфейса

```
interface Sum {
    int sum(int x, int y);
}
```

- Задаем Lambda-выражение:

```
Sum sm = (x, y)-> x+y;
```

- Вызываем Lambda-выражение:

```
int z = sm.sum(2, 3);
```

Чем хорошо применение lambda

- Код (поведение) можно оформить как переменные и передать в метод

Код как параметр метода

- Позволяет обращаться с кодом как с данными
  - Код (поведение) можно оформить как переменные и передать в метод

```
List<String> list = new ArrayList<String>();
list.add("first");
list.add("second");
list.add("third");
list.forEach((String s)->{System.out.println(s);});
```

Чем хорошо применение lambda

- Управление забирается у языка и передается библиотеке
- Нет необходимости менять язык – можно менять библиотеки

Внешнее итерирование

```
for (Shape s: shapes) {
    if (s.getColor() == RED)
        s.setColor(BLUE);
}
```

- Семантика цикла for привязана к языку
- Данный синтаксис прячет взаимодействие с библиотеками

Внутреннее итерирование

```
shapes.forEach(s->{
     if (s.getColor() == RED)
            s.setColor(BLUE);
   });
```

- Итерирование контролирует библиотека
- Библиотека может
  - распараллеливать,
  - итерировать в произвольном порядке,
  - применять “lazy”-подход
- Теперь проектировщики API могут «развернуться»

Эволюция интефейсов

- В интерфейсе Collection появился новый метод forEach

```
interface Collection<T> {
    forEach(Block<T> action) default {
        for (T t: this) {
            action.apply(t);
        }
    }
}
```

Default method – новое свойство языка. Виртуальный метод может иметь реализацию по умолчанию

Это позволяет передать управление библиотекам

Block<T>, Predicate<T> - функциональные интерфейсы в составе Java 8

Множественное наследование?

- Dafault-методы позволят добавить поведение к интерфейсу

- Означает ли это множественное наследование?
  - В Java всегда было наследование типов
  - Теперь есть наследование поведения
  - Но нет наследования состояния. И не будет.

  - Есть правила разрешения конфликтов

Collection, default-методы

- Java Collection Framework давно не менялся и с введением lambda стал устаревать
- Поэтому он был расширен

- Интерфейс Collection содержит новые методы:
  - forEach
  - removeAll
  - retailAll

- В наследниках можно заменить реализацию

```
interface Collection<T> {
    forEach(Block<T> action) default {
      for (T t: this) {
          action.apply(t);
      }
  }

    boolean removeAll default (
            Predicate <? Super T> filter) {
    boolean removed = false;
    Iterator<E> each = this.iterator();
    while(each.hasNext()) {
        if(filter.test(each.next())) {
            each.remove();
            removed = true;
        }
    }
    return removed;
  }
}
```

Bulk operations on Collections

- Групповые операции для коллекции:
  - filter (отфильтровать)
  - map (отобразить)
  - into (отобрать)

Bulk operations on Collections (cont.)

- Вернемся к примеру Shapes – применим lambda

```
shapes.forEach(s-> {
    if (s.getColor() == RED)
        s.setColor(BLUE);
}
```

```
shapes.filter(s->s.getColor()==RED)
      .forEach(s->{s.setColor(BLUE);});
```

Bulk operations on Collections (cont.)

- Отберем фигуры синего цвета в список

```
List<Shapes> blueBlocks =
    shapes.filter(s->s.getColor()==RED)
          .into(new ArrayList());
```

- Пусть каждая фигура находится в контейнере (Box), выберем те контейнеры, которые содержат фигуры синего цвета

```
Set<Box> hasBlueBlock =
    shapes.filter(s->s.getColor()==RED)
          .map (s->s.getContainingBox())
          .into(new HashSet<Box> ());
```

Bulk operations on Collections (cont.)

- Вычислим суму весов для синих фигур

```
int sumOfWeight = shapes
            .filter(s->s.getColor()==BLUE)
            .map(s->getWeight())
            .sum();
```

Преимущества групповых операций для коллекций

- Строим сложные операции из простых блоков
- Читаемый код
- Библиотеки могут применять
  - параллельные вычисления,
  - вычисления в произвольном порядке,
  - ленивые вычисления (например, найти первый элемент, который соответствует критерию)

Iterable

- Для введения групповых операций в Java 8 расширена абстракция Iterable

```
Iterable<Foo> fooIter = collection;
Iterable<Foo> filtered = fooIter.filter(f->f.isBlue());
Iterable<Foo> mapped = filtered.map(f->g.getBar());
mapped.forEach(s->System.out.println(s));
```

- Передача поведения в указанные методы возможна благодаря наличию в составе Java 8 функциональных интерфейсов Block<T>, Predicate<T>

Predicate

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

    Predicate<T> and(Predicate<? super T> p) default {
        return t -> this.test(t) && p.test(t);
    }

    Predicate<T> negate() default {
        return t -> !this.test(t);
    }

    Predicate<T> or(Predicate<? super T> p) default {
        return t -> this.test(t) || p.test(t);
    }

    Predicate<T> xor(Predicate<? super T> p) default {
        return t -> this.test(t) ^ p.test(t);
    }
}
```

Код приведен «схематически», см. JDK 8

Block

```
public interface Block<T> {
    void apply(T t);
}
```

Пример использования default-функций Predicate

```
Predicate<String> pr1 = f->f==null;
Predicate<String> pr2 = f->f.isEmpty();
Predicate<String> pr3 = pr1.or(pr2);

List<String> list = new ArrayList<String>();
list.add("Hello");
list.add("");
List<String> listF = new ArrayList<String>();
list.filter(pr3.negate()).into(listF);
```

Задача – отфильтровать непустые строки с помощью комплексного предиката.
Код с комплексными предикатами выглядит в Java-стиле

Iterable (cont.)

- filter
- map
- forEach
- into
- sorted
- aggregate (например, max, sum …)
- groupBy
- mapReduce

Пример

- Select author, sum(pages) from documents group by author

```
Map<String, Integer> map = new HashMap();
for (Document d: document) {
    String author = d.getAuthor();
    Integer i = map.get(author);
    if (i==null)
        i=0;
    map.put(author, i+d.getPageCount());
}
```

```
Map <String, Integer> map =
       documents.aggregateBy(d->d.getAuthor(),
                             ()->0,
                             (sum, d) ->sum+d.getPageCount());
```

Параллельное выполнение

- Цель: библиотеки должны прятать сложности реализации параллелизма
- Цель: сократить разрыв между синтаксисом последовательной и параллельной обработки
  - Сейчас синтаксисы абсолютно разные
  - Код для последовательной обработки выглядит просто
  - Код для параллельной обработки - сложно

Пример

```
int sumOfWeight = shapes
              .parallel()
              .filter(s->s.getColor()==BLUE)
              .map(s->getWeight())
              .sum();
```

Преимущества lambda

- С помощью lambda могут быть разработаны более выразительные API
- Некоторые аспекты функциональности могут быть внесены в поток управления выполнения, который контролируется библиотекой
  – Аналогия - своеобразный IoC на уровне языка и библиотек
- Больше возможностей для оптимизации
- Более читаемый код

Сортировка (Java-стиль)

```
Collections.sort(list, new Comparator() {
    public int compare(Person x, Person y)
        return x.getLastName().compareTo(y.getLastName());
})
```

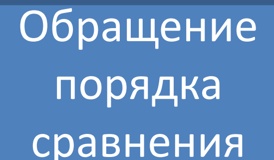
Default-методы - Combinators

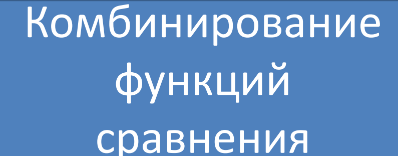
```
public interface Comparator<T> {

    int compare (T o1, T o2);

    Comparator<T> reverse () default {
        return (o1, o2)->compare (o2, o1);
    }

    Comparator<T> compose (Comparator<T> other) default {
        return (o1, o2) -> {
                int cmp = compare (o1, o2);
                return cmp!=0?cmp:other.compare(o1, o2);
        };
    }
}
```





Default-методы – Combinators (cont.)

```
Comparator<Person> byFirst = …
Comparator<Person> byLast = …

Comparator<Person> byFirstLast = byFirst.compose(byLast);
Comparator<Person> byLastDesc = byLast.reverse();
```

Сортировка с использованием lambda

```
class Comparators {
    public static <T, U extends Comparable<? Super U>>
    Comparator<T> comparing(Mapper <T, U> m) {
        return (x, y)->m.map(x).compareTo(m.map(y));
}
```

Использование:

```
Comparator<Person> byLastName =
       Comparators.comparing(p->p.getLastName());
Collections.sort(people, byLastName);
```

```
people.sort(Comparators.comparing(p->p.getLastName()))
people.sort(Comparators.comparing(p->p.getLastName()).reverse())
```

Задачи, которые пришлось решить при введении лямбда в Java

- Каков должен быть тип лямбда-выражений в Java?
  - В языке Java нет понятия функционального типа
  - В JVM нет презентации функционального типа в сигнатуре функции
- Требования:
  - Необходимо представлять функциональные типы в сигнатуре в JVM
  - Необходимо инстанцировать функциональные типы
  - При этом необходимо избежать значительных изменений в JVM

Литература

- http://tronicek.blogspot.com/2007/12/closures-closure-is-form-of-anonymous_28.html
- Brian Goetz, The Road to Lambda, https://oracleus.activeevents.com/connect/sessionDetail.ww?SESSION_ID=4862
- http://www.jcp.org/en/jsr/summary?id=335