Лямбда-выражения

- Java8, добавление поддержки функциональных программных конструкций
- Лямбда-выражения – компактный способ передать поведение из одного места программы в другое

Использование функциональных конструкций

- Компараторы
- Кнопки в GUI
- Потоки

Синтаксис

- (параметры) –> тело лямбда-выражения
- () –> System.out.println(″Example″)
- n –> {

  System.out.println(″Example″);

  System.out.println(n);

}

Использование лямбда-выражений

- Присвоение
- Передача в качестве аргумента функции

Пример лямбда-выражений (1)

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

Пример лямбда-выражений (2)

- public int compare(String firstStr, String secondStr)
  { return Integer.compare
  (firstStr.length(),secondStr.length());
  }
- (String firstStr, String secondStr) –>
  Integer.compare(firstStr.length(),
  secondStr.length());

Пример лямбда-выражений (3)

- Comparator<String> comp = (String firstStr, String secondStr) – > Integer.compare (firstStr.length(), secondStr.length());
- Comparator<String> comp = (firstStr, secondStr) –> Integer.compare(firstStr.length(),secondStr.length())

Контроль типов

- Тип результата лямбда-выражения никогда не указывается, это всегда выясняется из контекста
- Тип самого лямбда-выражения – функциональный интерфейс

Функциональные интерфейсы

- интерфейс, который содержит один абстрактный метод, который является типом лямбда-выражения
- метод сопоставится с лямбда-выражением при условии совместимости сигнатур

Предопределенные функциональные интерфейсы

- Interface Function<T,R>

R apply(T t)

- Выполняет операцию над объектом типа T и возвращает результат типа R

Предопределенные функциональные интерфейсы

- Interface Predicate<T>

boolean test(T t)

- Определяет удовлетворяети ли объект типа T условию

Предопределенные функциональные интерфейсы

- Interface UnaryOperator<T>

T apply(T t)

- Выполняет унарную операцию над объектом типа T и возвращает результат того же типа

Предопределенные функциональные интерфейсы

- Interface BinaryOperator<T>

T apply(T t, T u)

- Выполняет бинарную операцию над объектами типа T и возвращает результат того же типа

Интерфейс с единственным абстрактным методом

- Interface Comparable<T>

int compareTo(T o)

Методы интерфейсов в Java 8

- Абстрактные
- По умолчанию (default)
- Статические

Захват значений переменных

```
 final String text = getName();
public static void repeatText(int count) {
    Runnable r = () –> {
        for (int i = 0; i < count; i++) {
            System.out.println(text);
        }
    };
…
```

Ссылки на методы

Classname::methodName

- Ссылка на статический метод (ContainingClass::staticMethodName)
- Ссылка на метод конкретного объекта (ContainingObject::instanceMethodName)
- Ссылка на метод произвольного объекта конкретного типа (ContainingType::methodName)
- Ссылка на конструктор (ClassName::new)

Ссылка на статический метод

- (x, y) -> Math.pow(x, y)
- Math::pow

Ссылка на метод конкретного объекта

```
class ComparisonProvider {
    public int compareByName(Profile a, Profile b) {
        return a.getName().compareTo(b.getName());
    }

    public int compareByAge(Profile a, Profile b) {
        return a.getBirthday().compareTo(b.getBirthday());
    }
}
ComparisonProvider myComparisonProvider = new
                                                ComparisonProvider();
Arrays.sort(profilesAsArray, myComparisonProvider::compareByName);
```

Ссылка на метод произвольного объекта конкретного типа

```
String strs[] = {"a","c","B"};
Arrays.sort(strs, String::compareToIgnoreCase);
Arrays.sort(strs, (x, y) -> x.compareToIgnoreCase(y));
```

Ссылка на конструктор

```
public static <T, E extends Collection<T>, D extends Collection<T>>
    D transferElements (E sourceCollection,
                                Supplier<D> collectionFactory) {
        D result = collectionFactory.get();
        for (T t : sourceCollection) {
            result.add(t);
        }
        return result;
}
// profiles – коллекция из объектов класса Profile
Set<Profile> profileSetLambda = transferElements(profiles, () ->
                                                { return new HashSet<>(); });
Set<Profile> profileSet = transferElements(profiles, HashSet::new);
Set<Profile> profileSet = transferElements(profiles,
                                                    HashSet<Profiles>::new);
```

Java Stream API

Stream – это средство конструирования сложных операций над коллекциями с применением функционального подхода.

Пример:

```
Integer sumOddOld = 0;
        for(Integer i: collection) {
            if(i % 2 != 0) {
                sumOddOld += i;
            }
        }
Integer sumOdd = collection.stream().filter(o -> o % 2 != 0).
                                    reduce((s1, s2) -> s1 + s2).orElse(0);
```

Способы создания потоков

Создание потока из коллекции collection.stream()

```
Collection<String> collection;
Stream<String> streamFromCollection = collection.stream();
```

Создание потока из значений Stream.of(значение1,… значениеN)

```
Stream<String> streamFromValues = Stream.of("a1", "a2", "a3");
```

Создание потока из файла Files.lines(путь_к_файлу)

```
Stream<String> streamFromFiles =Files.lines(Paths.get("file.txt"));
```

Создание потока с помощью Stream.builder

```
Stream<String> streamFromBuilder =
  Stream.builder().add("a1").add("a2").add("a3").build();
```

Методы работы с потоками

Отложенные или конвейерные — возвращают другой stream, то есть работают как builder,

Терминальные или энергичные — возвращают другой объект, такой как коллекция, примитивы, объекты, Optional и т. д.

Общее правило: у потока может быть сколько угодно вызовов конвейерных вызовов и в конце один терминальный, при этом пока не будет вызван терминальный метод никаких действий не происходит.

Отложенные методы (1)

Возвращает поток, состоящий из элементов исходного потока, соответствующих условию:

Stream<T> **filter**(Predicate<? super T> predicate)

```
collection.stream().filter(o -> o % 2 != 0)
```

Возвращает поток, состоящий из преобразованных элементов исходного потока:

<R> Stream<R> **map**(Function<? super T,? extends R> mapper)

```
collection.stream().map((s) -> s + "_1")
```

Не изменяет исходный поток, а применяет указанные действия к каждому элементу потока:

Stream<T> **peek**(Consumer<? super T> action)

```
collection.stream().peek((e) -> System.out.print(", " + e))
```

Отложенные методы (2)

Сортируют поток:

Stream<T> **sorted**()

Stream<T> **sorted**(Comparator<? super T> comparator)

Ограничивает поток первыми элементами в соответствии с заданным количеством:

Stream<T> **limit**(long maxSize)

Откидывает n первых элементов потока:

Stream<T> **skip**(long n)

Терминальные методы (1)

Построение некоторого конечного значения:

<R,A> R **collect**(Collector<? super T,A,R> collector)

Пример:

```
collection.stream().map(String::toUpperCase).peek((e) ->
   System.out.print("," + e)).collect(Collectors.toList());
```

void **forEach**(Consumer<? super T> action)

Пример:

```
collection.stream().forEach((e) -> System.out.print(", " + e))
```

long **count**()

Пример:

```
long k =
   collection.stream().filter(«a1»::equals).count();
```

Терминальные методы (2)

boolean **anyMatch**(Predicate<? super T> predicate)

boolean **allMatch**(Predicate<? super T> predicate)

boolean **noneMatch**(Predicate<? super T> predicate)

Optional<T> **findFirst**()

Optional<T> **max**(Comparator<? super T> comparator)

Optional<T> **min**(Comparator<? super T> comparator)

Optional<T> **reduce**(BinaryOperator<T> accumulator)

T **reduce**(T identity, BinaryOperator<T> accumulator)

Класс Optional<T>

T get()

boolean isPresent()

T orElse(T other)

Optional<T> filter(Predicate<? super T> predicate)

void ifPresent(Consumer<? super T> consumer)