
Java™

ORACLE

Project Lambda в JDK8

Дмитрий Козорез
dmitry.kozorez@oracle.com

ORACLE

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.

The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

ORACLE

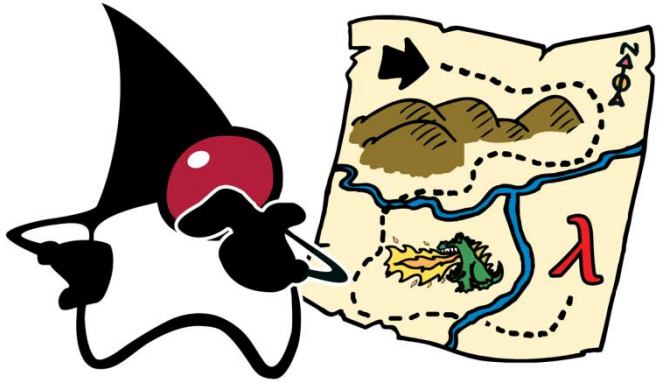
То что вы услышите далее, является непрерывным потоком сознания, который я буду стараться дифференцировать хотя бы по интонации.

Но, как показывает практика, а тем кто знаком с теорией, и она, непрерывность, к сожалению, не гарантирует даже существования частных производных.

«Не вынесла душа поэта
дифференцировать по η"
А.И. Назаров

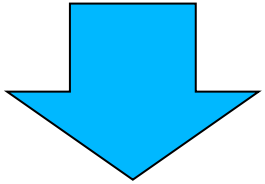
Развивая Java

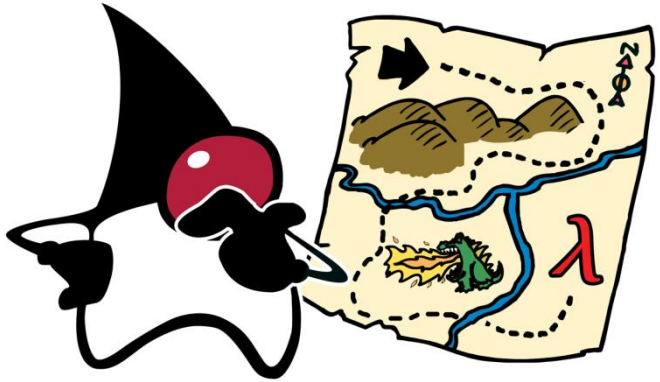
Lambda expressions(closures, замыкания)
Default methods (развитие интерфейсов)



Развивая Java

Lambda expressions(closures, замыкания)
Default methods (развитие интерфейсов)





Java SE 8 большой шаг в развитии библиотек Java
- Групповые операции на Collections
- «Параллельные» библиотеки

Что же такое Lambda Expression?

Lamba expression(closure) = анонимный метод

- который имеет список аргументов, возвращаемый тип и тело

```
(Object o) → o.toString()
```

Что же такое Lambda Expression?

Lamba expression(closure) = анонимный метод

- который имеет список аргументов, возвращаемый тип и тело

```
(Object o) → o.toString()
```

- может использовать локальные переменные

```
(Person p) → p.getName().equals(name)
```

Method reference это ссылка на существующий метод

```
Object::toString
```

Времена изменились

В 1995 большинство популярных языков не поддерживали замыкания
Сегодня, Java один из последних распространенных язков, не имеющих поддержки замыканий.

- В C++ они были недавно добавлены
- В C# с версии 3.0
- Все создаваемые ныне языки создаются с поддержкой замыканий

“In another 30 years people will laugh at anyone who tries to invent a language without closures, just as they’ll laugh now at anyone who tries to invent a language without recursion.”

Mark Jason Dominus, The Perl Review (April 7, 2005)

Длинная дорога Java к замыканиям

1997 – Odersky/Wadler экспериментальный проект “Pizza”

1997 – Java 1.1 добавление вложенных классов – слабой формы замыканий

– слишком сложные правила разрешения имен и много ограничений

2006-2008 многочисленные дебаты в сообществе о замыканиях

– никаких существенных сдвигов

Длинная дорога Java к замыканиям

Декабрь 2009 – OpenJDK Project Lambda
Ноябрь 2010 – JSR-335 создан
Текущий статус

- Спецификация EDR (early draft review) готова
- Прототип (исходный код и бинарные файлы) доступны в OpenJDK
- Часть Java SE 8

External Iteration

Пример перекрашивает красные блоки в синий цвет

Используя foreach цикл

- Цикл последователен
- Клиент вынужден управлять итерациями

Это называется external iteration

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

Lambda Expressions

Переписанный с использованием lambda и Collection.forEach

- Не просто синтаксический сахар
- Теперь библиотека контролирует проход по циклу
- Это внутренняя итерация
- Больше “что”, меньше “как”

```
Shapes.forEach( s →  {
     if (s.getColor() == RED)
          s.setColor(BLUE);
})
```

Функциональные интерфейсы

Какой тип у лямбды или ссылки на метод?

- Лямбда = интерфейс с одним методом*
- Runnable, Comparator, ActionListener
- functional interfaces
- Predicate<T>, Block<T>

Лямбда рассматривается как экземпляр функционального интерфейса

```
Predicate<String> isEmpty = s → s.isEmpty();
Predicate<String> isEmpty = String::isEmpty;
Runnable r = () → {System.out.println("Boo!") };
```

Эволюция интерфейсов

Collection.forEach()

Эволюция интерфейсов

Collection.forEach() = виртуальный метод с реализацией по умолчанию
Библиотекам надо развиваться!

Эволюция интерфейсов

Collection.forEach() = виртуальный метод с реализацией по умолчанию
Библиотекам надо развиваться!

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

Эволюция интерфейсов

Интерфейсы – палка о двух концах

- Невозможно изменить пока вы не контролируете все реализации
- Реальность: возраст API
  - С добавлением новых фич, существующий API выглядит еще старше
- Множество проблем со старым API
  - Нельзя позволять API застаиваться
  - Полностью заменять (хоть каждые несколько лет!)

Накладные расходы по развитию API должны быть на реализаторах, а не на пользователях

Методы по умолчанию(default methods)

Множественное наследование в Java?

“В действительности всё совсем не так, как на самом деле.”

Станислав Ежи Лец

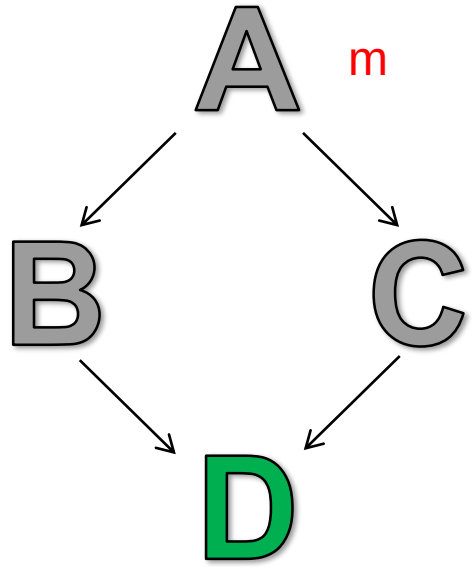
Правила наследования

- Правило 1: класс побеждает
- Правило 2: более специфичный интерфейс предпочтительней
- Правило 3: во всех остальных случаях пользователь должен предоставить реализацию

Diamonds? Нет проблем!

Для D существует единственный наиболее специфичный метод по умолчанию из интерфейса A

- D наследует m() из A

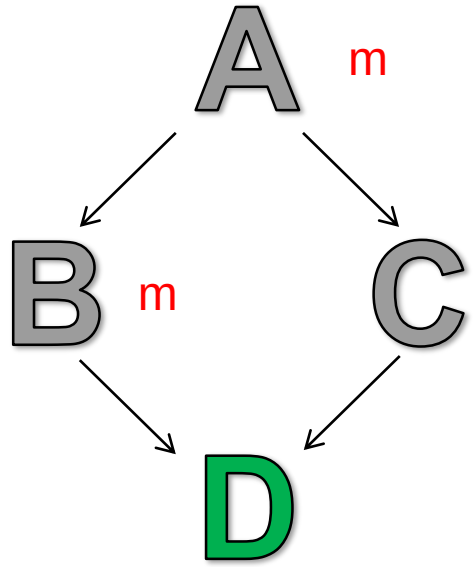



```
interface A {
     default void m() {…}
}
interface B extends A {…}
interface C extends A {…}
class D implements B, C {...}
```

Diamonds? Нет проблем!

Если B предоставит дефолтную реализацию

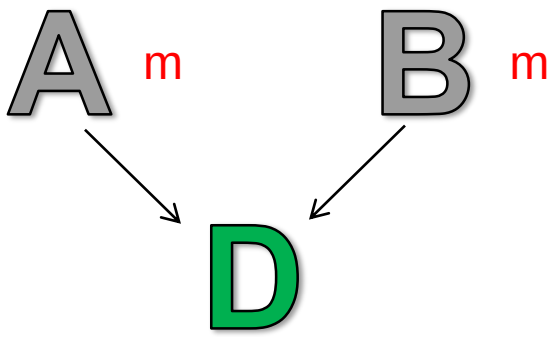
- B более специфичен, чем A
- D наследует m() из B




```
interface A {
     default void m() {…}
}
interface B extends A {
     default void m() {…}
}
interface C extends A {...}
class D implements B, C {...}
```

Явное разрешение неоднозначности

Если правила 1 и 2 не применимы, подкласс должен реализовать метод




```
interface A {
    default void m() {…}
}
interface B {
    default void m() {…}
}

class C implements A, B {
//Must implement/reabstract m()
    void m() { A.super.m(); }
}
```

Default methods – развивая интерфейсы

Новые методы Collection
- forEach(Block)
- removeIf(Predicate)

Подклассы могут предоставить лучшую реализацию

```
interface Collection<E> {
     default void forEach(Block<E> action) {
          for (E e: this)
               action.apply(e);
     }

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

Default Methods в кач-ве опциональных

Методы по умолчанию могут снизить накладные расходы по реализации
Большинство реализаций Iterator не предоставляют полезного remove()

```
interface Iterator<T> {
boolean hasNext();

}
```

Default Methods в кач-ве опциональных

Методы по умолчанию могут снизить накладные расходы по реализации
Большинство реализаций Iterator не предоставляют полезного remove()

```
interface Iterator<T> {
boolean hasNext();

T next();
default void remove() {
     throw new UnsupportedOperationException();
     }
}
```

Методы по умолчанию

Comparator.reverse() – “разворачивает” Comparator
Comparator.compose() – рассчитывает лексикографический порядок с использованием двух компараторов

```
interface Comparator<T> {
int compare(T o1, T o2);
    default Comparator reverse() {
        return (o1, o2) → compare(o2, o1);
    }

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

Bulk operations на коллекциях

Групповые операции: filter, map, into, …
“Покрасьте красные блоки в синий” = filter+forEach

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

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

Bulk operations на коллекциях

Собрать все синие фигуры в List

```
List<Shape> blueBlocks = shapes
.stream().filter(s → s.getColor() == BLUE)
.into(new ArrayList<>());
```

Если фигуры находятся в Box, то найдем содержащие синие фигуры

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

Bulk operations на коллекциях

Посчитать суммарный вес синих фигур

```
Int sumOfWeight = shapes.stream()
.filter(s → s.getColor() == BLUE)
.map(s → s.getWeight())
.sum();
```

Или используя method references

```
Int sumOfWeight = shapes.stream()
.filter(s → s.getColor() == BLUE)
.map(Shape::getWeight)
.sum();
```

Bulk operations на коллекциях

Новые групповые операции выразительны и компонуемы

– Собрать сложную операцию из простых блоков
– Каждый шаг посвящен одному действию
– Структура клиентского кода менее хрупкая
– Меньше промежуточных структур
– Библиотека инкапсулирует операцию: параллелизм, смешанный порядок, производительность или ленивость

Laziness

Когда происходит расчет?
Итераторы ленивы
Ленивость может быть эффективна, когда не нужен весь результат

Потоки (Streams)

Для того чтобы добавить групповые операции, мы создали новую абстракцию -- Stream

- Представляет собой поток значений
  - Не является структурой данных – не хранит значения
- Источником данных может быть Collection, массив, генерирующая функция, I/O...
- Операции, производящие новые потоки, ленивы

```
Stream<Foo> fooStream = collection.stream();
Stream<Foo> filtered = fooStream.filter(f → f.isBlue());
Stream<Bar> mapped = filtered.map(f → getBar());
```

Потоки (Streams)

Для того чтобы добавить групповые операции, мы создали новую абстракцию -- Stream

- Представляет собой поток значений
  - Не является структурой данных – не хранит значения
- Источником данных может быть Collection, массив, генерирующая функция, I/O...
- Операции, производящие новые потоки, ленивы
- Способствуют более “беглому” стилю использования

```
collection.stream()
.filter(f → f.isBlue())
.map(f → f.getBar())
.forEach(System.out::println);
```

Потоки (Streams)

Операции делятся на промежуточные (intermediate) и терминальные (terminal)

- Вычислений не происходит пока не вызвана терминальная операция
- Библиотека выбирает стратегию реализации

```
Interface Stream<T> {
    …
    Stream<T> filter(Predicate<T> predicate);
    <U> Stream<U> map(Mapper <T, U> mapper);

    Stream<T> uniqueElements();
    Stream<T> sorted(Comparator<? super T> cmp);
    Stream<T> limit(int n);
    Stream<T> skip(int n);
    Stream<T> concat(Stream<? extends T > other);
    …
}
```

Потоки (Streams)

Операции делятся на промежуточные (intermediate) и терминальные (terminal)

- Вычислений не происходит пока не вызвана терминальная операция
- Библиотека выбирает стратегию реализации

```
interface Stream<T> {
    …
    void forEach(Block<? super T> block);
    T[] toArray(Class<T> classToken);
    T aggregate(T base, BinaryOpertor<T> operator);
    Optional<T> findFirst();
    Optional<T> findAny();
    …
}
```

Потоки и Laziness

Потоки это ленивые коллекции – расчет происходит когда элементы нужны

- Хорошая экономия ресурсов в случаях вроде filter-map-findFirst
- Laziness почти незаметна (никаких LazyList, LazySet, etc)

Параллелизм

Простые для использования параллельные библиотеки для Java

Уменьшить концептуальную и синтаксическую разницу между параллельными и последовательным выражениями тех же расчетов

Сделать параллелизм явным, но ненавязчивым

Параллельное суммирование с коллекциями

Параллельное суммирование с групповыми операциями на коллекциях

```
int sumOfWeight =
shapes.parallel().filter(s → s.getColor() == BLUE)
     .map(Shape::getWeight)
     .sum();
```

- Явный, но ненавязчивый параллелизм
- Все три операциии слиты в один параллельный проход

Параллельные потоки

Потоки могут обрабатываться как параллельно, так и последовательно

- В зависимости от источника потока
  - Collection.stream()
  - Collection.parallel()

Некоторые структуры данных дробятся лучше других

- Массивы дробятся лучше чем списки
- Структура данных вносит стратегию дробления
- Параллельные операции над потоком могут быть запущены на любой дробимой структре данных

Дробление

Параллельный аналог Iterator это... Spliterator

- Определяет стратегию дробления для структуры данных

```
public interface Spliterator<T> {
    int getNaturalSplits();

    Spliterator<T> split();
    Iterator<T> iterator();

    default int getSizeIfKnown();
    default int estimateSize();
    default boolean isPredictableSplits();
}
```

Lambda позволяют улучшить API

Лямбды позволяют предоставить более мощный API

Граница между библиотекой и клиентом более проницаема

- Клиент может предоставить функциональность, которую можно добавить к исполнению
- Клиент решает что делать, библиотека решает как это сделать
- Безопаснее, предоставляет больше возможностей для оптимизации

Пример: сортировка

Если необходимо отсортировать список, сегодня вы напишете Comparator

- Компаратор объединяет извлечение поискового ключа и упорядочивание по нему

Можно переписать с использованием лямбд без особых трудностей

- В качестве бонуса получает разделение упомянутых аспектов

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

Пример: сортировка

Лямбды позволяют лучше разделять API

- Добавим метод, который принимает “извлекатель ключа” и возвращает Comparator
- Сразу много версий: для Object, int, long, etc

```
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));
    }
}
```

Пример: сортировка

Лямбды позволяют лучше разделять API

- Добавим метод, который принимает “извлекатель ключа” и возвращает Comparator
- Сразу много версий: для object, int, long, etc

```
- Comparator<Person> byLastName
=Comparators.comparing(p → p.getLastName());
```

```
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));
    }
}
```

Пример: сортировка

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

Можно еще улучшить:

```
Collections.sort(people, comparing(p → p.getLastName()));
```

И еще:

```
people.sort(comparing(p → p.getLastName()))
```

И еще!

```
people.sort(comparing(Person::getLastName))
```

Пример: сортировка

А если вдруг надо в обратном порядке?

```
people.sort(comparing(Person::getLastName).reverse());
```

А если еще и по имени?

```
people.sort(comparing(Person::getLastName)
        .compose(comparing(Person::getFirstName)));
```

Lambda позволяют улучшить API

Метод comparing() один из построенных для лямбд

- Принимает функцию-“экстрактор” и возвращает функцию-”компаратор”
- Выделяет функцию извлечения ключа из сравнения
- Улучшает переиспользуемость

Ключевое воздействие на API: больше возможностей для компоновки

Лямбды в языке:

- Позволяют писать библиотеки лучше
- Читаемее, код менее подвержен ошибкам

Lambda позволяют улучшить API

Мы предпочитаем развивать Java через развитие библиотек

Но иногда, мы достигаем пределов практической выразимости в библиотеках, и нуждаемся в небольших изменениях в языке

Небольшие изменения в языке в нужном месте могут дать потрясающий результат!

Итоги: зачем это нужно?

λ + default methods = эволюция интерфейсов

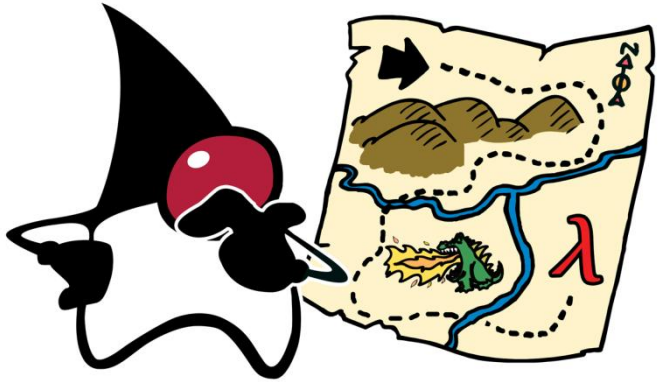
(в частности, bulk operations)

И где же мы ныне? Прогресс

EDR spec доступен на http://openjdk.java.net/projects/lambda
RI Binary доступны по адресу http://jdk8.java.net/lambda

Скачайте и попробуйте сами!

Лямбды войдут в Java SE 8



MAKE THE FUTURE JAVA

ORACLE®