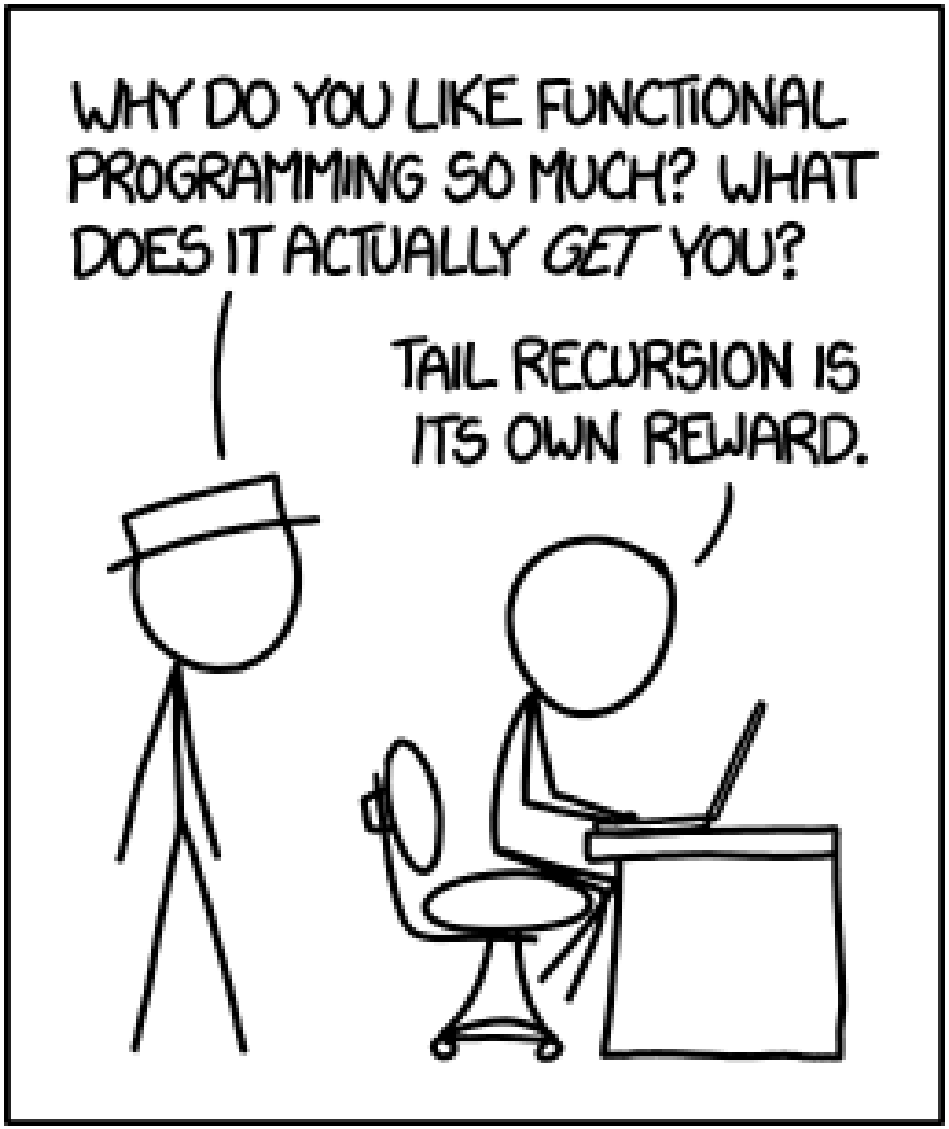
Элементы функционального программирования

Алексей Владыкин

20 октября 2014



https://xkcd.com/1270/

Функциональные языки

- LISP + диалекты
- Erlang
- F#
- Haskell
- и другие

- Процесс вычисления представляет собой вычисление значения функции (в математическом смысле)
- Функции высших порядков
- λ-выражения
- Ленивые вычисления


КАКОЕ ОТНОШЕНИЕ
ВСЕ ЭТО ИМЕЕТ К JAVA?
memegenerator.net

- Функции представляются объектами
- Один из методов объекта вычисляет значение функции
- Интерфейсы для наиболее распространенных функций собраны в пакете `java.util.function`
- Такие интерфейсы с единственным абстрактным методом называются функциональными и помечаются аннотацией `@FunctionalInterface`

```
package java.util.function;

@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    default Predicate<T> negate() {
        // ...
    }

    // and, or
}
```

```
package java.util.function;

@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    default <V> Function<V, R> compose(
        Function<? super V, ? extends T> before) {
        // ...
    }

    // andThen
}
```

```
package java.util.function;

@FunctionalInterface
public interface Supplier<T> {

    T get();
}
```

```
package java.util.function;

@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

    default Consumer<T> andThen(
            Consumer<? super T> after) {
        // ...
    }
}
```

```
package java.util;

@FunctionalInterface
public interface Comparator<T> {

    int compare(T o1, T o2);

    // ...
}
```

- `find(Predicate<T> predicate)`
- `sort(Comparator<T> order)`
- `forEach(Consumer<T> action)`
- `map(Function<T, R> func)`

```
List<User> users = getAllUsers();
Collections.sort(users, new Comparator<User>() {
    @Override
    public int compare(User u1, User u2) {
        return u1.getName()
                .compareTo(u2.getName());
    }
});
```

- Новый компактный синтаксис для инстанцирования функциональных интерфейсов

```
List<User> users = getAllUsers();
Collections.sort(users, (u1, u2) ->
        u1.getName().compareTo(u2.getName()));
```

- Компилятор сам выводит типы
- Фигурные скобки не нужны, если внутри одно выражение

```
List<User> users = getAllUsers();
Collections.sort(users,
        Comparator.comparing((u) -> u.getName()));
```

Iterable.forEach()

```
List<User> users = getAllUsers();
list.forEach((u) ->
        System.out.println(u.getName()));
```

Collection.removeIf()

```
void removeUserByName(String userName) {
    List<User> users = getAllUsers();
    users.removeIf((u) ->
            u.getName().equals(userName));
}
```

- Можно обращаться к тем локальным переменным, которые «effectively final»

List.replaceAll()

```
List<String> list = getList();
list.replaceAll((s) -> {
    StringBuilder sb = new StringBuilder(s);
    return sb.reverse().toString();
});
```

- Еще один способ инстанцирования функциональных интерфейсов: по имени метода

```
List<Integer> list = getList();
list.forEach(System.out::println);
```

```
List<User> users = getAllUsers();
Collections.sort(users,
        Comparator.comparing(User::getName));
```

```
public void example() {
    List<String> names = Arrays.asList(
            "John", "Frank", "Sam");
    List<User> users = map(names, User::new);
}

public <S, T> List<T> map(
        List<S> list,
        Function<S, T> function) {
    List<T> result = new ArrayList<T>(list.size());
    list.forEach((s) ->
            result.add(function.apply(s)));
    return result;
}
```


Chuck Norris

java.util.stream

- Stream — последовательность элементов (возможно, бесконечная) с поддержкой различных преобразований

```
List<User> users = getAllUsers();
users.stream()
     .filter((u) -> u.getAge() > 20)
     .sorted(Comparator.comparing(User::getName))
     .limit(3)
     .forEach(System.out::println);
```

- Источник → промежуточные операции → терминальная операция

Источники Stream'ов

- `Collection.stream()`
- `Files.walk(Path)`, `Files.list(Path)`
- `BufferedReader.lines()`
- `Stream.iterate(T, UnaryOperator<T>)`, `Stream.generate(Supplier<T>)`, `Stream.of(T...)`

Промежуточные операции

- `filter(Predicate<T>)`
- `map(Function<T,R>)`
- `flatMap(Function<T,Stream<R>>)`
- `peek(Consumer<T>)`
- `sorted(Comparator<T>)`
- `distinct()`
- `limit(long)`
- `skip(long)`

Терминальные операции

- `forEach(Consumer<T>)`
- `findFirst()`, `findAny()`
- `allMatch(Predicate<T>)`, `anyMatch(Predicate<T>)`, `noneMatch(Predicate<T>)`
- `reduce(BinaryOperator<T>)`
- `collect(Collector<T>)`
- `min(Comparator<T>)`, `max(Comparator<T>)`

Пример

```
private static class FibonacciSupplier
        implements Supplier<BigDecimal> {

    private BigDecimal prev = BigDecimal.ZERO;
    private BigDecimal next = BigDecimal.ONE;

    @Override
    public BigDecimal get() {
        BigDecimal current = next;
        next = prev.add(current);
        prev = current;
        return current;
    }
}
```

Пример

```
BigDecimal million = new BigDecimal(1_000_000);
Stream.generate(new FibonacciSupplier())
        .filter((d) -> d.compareTo(million) >= 0)
        .limit(1)
        .forEach(System.out::println);
```

Что сегодня узнали

- Что такое функциональное программирование
- Какие средства добавлены в Java 8 для поддержки ФП
- Как писать на Java в функциональном стиле