Лекция 1
Разработка АС реального времени (часть 2)

ФИО преподавателя: Зорина Наталья Валентиновна

e-mail: zorina_n@mail.ru

Тема лекции:

«Функциональные интерфейсы и потоки. STREAM API»

Функциональное программирование

Стиль программирования, основные черты:

1. Предпочтение чистых функций (отсутствие побочных эффектов)
2. Функции как объекты первого класса (first-class citizen):
    1. Могут храниться в переменных
    2. Могут передаваться как параметры
    3. Могут возвращаться как результат

Функциональное программирование

Языки, ориентированные на функциональное программирование:

1. Lisp family: Common Lisp, Scheme, Clojure (1960)
2. ML family: Standard ML, OCaml, F# (1975)
3. Erlang (1986)
4. Haskell (1990)

Функциональное программирование

В 2000-е годы элементы функциональных языков начали проникать в мэйнстримные императивные языки:

1. Лямбда-выражения: практически все языки
2. Списочные выражения (list comprehensions): Python
3. Алгебраические типы и pattern matching (C#, Rust, Swift, Java 18?)

Функциональное программирование

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T arg);
}
```

В Java функциями являются объекты, реализующие функциональные интерфейсы – интерфейсы с только одним абстрактным методом.

Лямбда-выражения:

```
Function<String, Integer> f = str -> str.length();
```

Функциональное программирование

Лямбда-выражения:

```
Function<String, Integer> f = str -> str.length();
```

Аналог в Java до версии 8:

```
Function<String, Integer> f = new Function<String, Integer>() {
    @Override
    public Integer apply(String str) {
        return str.length();
    }
};
```

Функциональное программирование

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T arg);
}
```

Аннотация @FunctionalInterface необязательная. Компилятор проверяет для интерфейсов, помеченных этой аннотацией, что в нем только один абстрактный метод.

Функциональное программирование

Модуль A	Модуль B

Версия 1:
interface UserDetails {
String getUserName();
}
	UserDetails unknown = () -> “unknown”;

Версия 2:
interface UserDetails {
String getUserName();
String getUserAddress();
}
	Ошибка компиляции:
	UserDetails unknown = () -> “unknown”;
	UserDetails больше не функциональный интерфейс! Нужно использовать анонимный класс.

Аннотация @FunctionalInterface используется, чтобы разработчики модуля A случайно не сломали зависящие от него модули. В случае ее наличия ошибки компиляции будут в модуле A.

Функциональное программирование

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T arg);
    default <V> Function<V, R> compose(
            Function<? super V, ? extends T> before) {
        return (V v) -> apply(before.apply(v));
    }
}
```

Методы по умолчанию могут присутствовать в функциональных интерфейсах.

Функциональное программирование

```
Function<String, Integer> getLength = str -> str.length();
// Также можно написать getLength = String::length;
Function<Integer, Double> half = n -> n / 2.0;
Function<String, Double> f = half.compose(getLength);
```

compose – функция высшего порядка: принимает функцию как параметр и возвращает функцию.

Функциональное программирование

Функции можно хранить в полях класса и использовать для параметризации поведения объектов (паттерн “Стратегия”).

Пример: вместо иерархии наследования можно передавать поведение как параметр.

class Animal { **abstract** String talk(); }

class Dog **extends** Animal { String talk() { **return** “Гав”; }

class Cat **extends** Animal { String talk() { **return** “Мяу”; }

Функциональное программирование

```
class Animal {
    private Supplier<String> talk;
    // Конструктор
    ... System.out.println(talk.get());
}
Animal dog = new Animal(() -> “Гав”);
Animal cat = new Animal(() -> “Мяу”);
```

Функциональное программирование

Побочные эффекты функции – то, что делает функция кроме вычисления результата функции.

Наиболее частый побочный эффект – изменение состояния объектов (т.е. изменение значений полей).

Ввод/вывод также является побочным эффектом.

Часто повторный вызов функции с побочным эффектом производит результат, отличный от предыдущего вызова.

Функциональное программирование

```
class Point {
    int x;
    int y;
    void move(int dx) {
        this.x += dx; // побочный эффект
    }
}
Point p1 = new Point(0, 0);
p1.move(3); // p1.x == 3
p1.move(3); // p1.x == 6
```

Функциональное программирование

Чистая (pure) функция – функция без побочных эффектов, результат которой зависит только от значений ее параметров.

```
class Point {
   int x;
   int y;
   void move(int dx) {
      this.x += dx; // побочный эффект
   }
   int getX() { return x; } // не является чистой, т.к. разные вызовы
                            // могут вернуть разные значения
}
```

Функциональное программирование

```
class Point {
    private int x;
    private int y;
    Point(int x, int y) { this.x = x; this.y = y; }
    Point move(int dx) {
        return new Point(this.x + dx, y); // нет побочных эффектов
    }
    int getX() { return x; }
}
```

И move, и getX – чистые функции (в данном случае this мы считаем неявным параметром функции).

Функциональное программирование

В ООП чистота функций (методов) достигается с помощью использования неизменяемых (immutable) классов. Все методы неизменяемого класса являются чистыми. После создания объекта неизменяемого класса его состояние больше не меняется. При операциях с такими объектами создаются новые объекты, а не меняются существующие.

Для того, чтобы гарантировать неизменяемость свойств объекта, следует использовать модификатор **final** для полей. Также неизменяемый класс как правило закрыт для наследования, т.к. иначе класс-наследник может добавить изменяемые поля.

Функциональное программирование

Пример описания неизменяемого класса:

```
public final class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Функциональное программирование

Этого недостаточно – все поля класса должны также быть неизменяемыми или недоступными извне:

```
public final class Mutable {
    private final int[] arr;
    public Mutable(int[] arr) { this.arr = arr; }
}
int[] array = {1, 2, 3};
Mutable mut = new Mutable(array);
array[1] = 20; // Состояние mut изменилось!
```

Функциональное программирование

Плюсы неизменяемых объектов:

1. Проще понять работу класса
    1. Отсутствует “действие на расстоянии”
    2. Результат работы программы не зависит от истории
    3. Проще проверять инварианты класса
2. Проще тестировать класс
3. Проще использовать в многопоточной среде
4. Можно использовать в качестве ключей Map

Функциональное программирование

Действие на расстоянии:

```
class Point {
    int x;
    int y;
}
Point p1 = new Point(0, 0);
Point p2 = p1;
...
p2.x = 10; // p1.x меняется тоже!
```

Функциональное программирование

Так как действие на расстоянии по определению нелокально, то его невозможно отследить, рассматривая только один изолированный участок кода.

Иногда действие на расстоянии полезно, на чаще нет.

Функциональное программирование

Зависимость от истории:

```
class MyClass {
    private ImportantThing thing;
    void init(ImportantThing thing) {
        this.thing = thing;
    }
    void work() {
        thing.doWork();
    }
}
```

Функциональное программирование

Зависимость от истории:

```
MyClass obj = new MyClass();
obj.init(new ImportantThing());
// Если мы забудем вызов init, то здесь будет ошибка:
obj.work();
```

Т.е. работоспособность метода зависит от того, какие методы мы вызывали перед этим.

Функциональное программирование

Для неизменяемого класса мы не можем забыть инициализацию:

```
class MyClass {
    private final ImportantThing thing;
    MyClass(ImportantThing thing) {
        this.thing = thing;
    }
    void work() {
        importantThing.doWork();
    }
}
```

Функциональное программирование

Благодаря отсутствию зависимости от истории вызванных методов неизменяемые классы проще тестировать – не нужно перед вызовом тестируемого метода вызывать методы, которые приводят объект в нужное состояние – он сразу инициализирован в правильном состоянии.

Функциональное программирование

Инвариант класса – условие, которое должно выполняться на протяжении всего срока жизни объекта.

В случае изменяемого объекта инвариант нужно проверять в каждом методе, изменяющем состояние (mutator method).

Это можно забыть сделать, и инвариант будет нарушен.

В неизменяемом классе инвариант достаточно проверять в конструкторе.

Функциональное программирование

```
// Инвариант: a < b
class OrderedPair {
    private int a;
    private int b;

    OrderedPair(int a, int b) {
        check(a < b);
        this.a = a;
        this.b = b;
    }
```

Функциональное программирование

```
class OrderedPair {
    ...
    void setA(int newA) {
        check(newA < b);
        this.a = newA;
    }
    void setB(int newB) {
        check(a < newB);
        this.b = newB;
    }
```

Функциональное программирование

В языке Java модификатор **final** для полей имеет особый смысл при многопоточной работе: к полю с этим модификатором могут безопасно обращаться одновременно несколько потоков, и они гарантированно будут видеть одинаковое значение.

Для не-final полей без особой синхронизации потоков может быть так, что каждый поток видит свое значение поля.

Функциональное программирование

Многие стандартные классы являются неизменяемыми:

1. String
2. Byte/Short/Integer/Long/Character/Boolean/Double
3. java.util.Optional
4. Классы пакета java.time
5. java.nio.file.Path
6. java.nio.charset.Charset
7. java.net.URI

Функциональное программирование

В Java начиная с версии 16 добавлена возможность краткой записи простых неизменяемых классов:

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
    public int x() {
        return x;
    }
    public int y() {
        return y;
    }
}
```

```
public record Point(int x, int y) {
}
```

Кроме того, объявление record определяет методы equals, hashCode и toString с использованием значений полей класса (их можно переопределить вручную).

Функциональное программирование

Можно ли программировать с использованием только неизменяемых объектов? Пример чисто функциональных языков типа Haskell показывает, что можно. Но это не всегда удобно:

1. Некоторые алгоритмы проще формулируются с использованием изменяемых структур данных
2. Так как любая операция с неизменяемыми объектами создает новый объект, это сильнее загружает сборщик мусора
3. Многие алгоритмы и структуры данных работают более эффективно на основе изменяемых объектов (массив – базовая структура данных – это изменяемый объект)

Функциональное программирование

Еще одно неудобство – необходимость создавать объект в “готовом” состоянии через конструктор. Для сложных объектов может понадобиться передать в конструктор много параметров, а в Java значений по умолчанию нет:

```
class MyHttpServer {
    final int port;
    final String protocol;
    final Path rootDir;
    MyHttpServer(int port, String protocol, Path rootDir) {
        ...
    }
```

Функциональное программирование

MyServer server = new MyServer(80, “http”, Path.of(“web”));

Мы хотим, чтобы эти параметры использовались по умолчанию!

Паттерн Builder позволяет использовать изменяемый объект для задания свойств, а затем создания на их основе неизменяемого объекта.

Как правило изменяемый Builder существует в рамках одного метода и снаружи не виден, так что это не нарушает функциональной чистоты.

Функциональное программирование

```
class MyServerBuilder {
    private int port = 80;
    private String protocol = “http”;
    private Path rootDir = Path.of(“web”);
    void setPort(int port) { this.port = port; }
    void setProtocol(String protocol) { this.protocol = protocol; }
    void setRootDir(Path rootDir) { this.rootDir = rootDir; }
    MyServer build() {
        return new MyServer(port, protocol, rootDir);
    }
}
```

Функциональное программирование

Использование:

```
MyServerBuilder builder = new MyServerBuilder();
builder.setPort(8080);
MyServer server = builder.build();
```

Функциональное программирование

Вариация этого подхода – Fluent Builder, где каждый setter возвращает this:

```
class MyServerBuilder {
    MyServerBuilder setPort(int port) {
        this.port = port;
        return this;
    }
    ...
}
MyServer server = new MyServerBuilder()
    .setPort(8080)
    .build();
```

Функциональное программирование

Стандартные функциональные интерфейсы и их методы:

1. Function: R apply(T arg)
2. Supplier: T get()
3. Consumer: void accept(T arg)
4. Predicate: boolean test(T arg)
5. BiFunction: R apply(T arg1, U arg2)
6. BiConsumer: void accept(T arg1, U arg2)
7. BiPredicate: boolean test(T arg1, U arg2)

Также есть варианты этих интерфейсов для примитивных типов int, long и double.

Функциональное программирование

В своих программах имеет смысл определять свои функциональные интерфейсы, а не пользоваться предопределенными, потому что:

1. Название интерфейса и метода лучше документирует его назначение
2. Можно использовать конкретные типы вместо типовых параметров, что облегчает чтение
3. В среде разработки будет проще найти реализации этого интерфейса
4. Стандартные интерфейсы не предусматривают использование checked exceptions

Функциональное программирование

Функциональные интерфейсы используются во многих методах библиотеки коллекций:

V Map.computeIfAbsent(K key, Function<K, V> create);

Для лямбда-выражений, вызывающих конструктор, есть краткая форма, по аналогии с методами:

```
Function<Integer, String[]> newArray = n -> new String[n];
Function<String, StringBuilder> toBuf = str -> new StringBuilder(str);
```

можно записать как

```
Function<Integer, String[]> newArray = String[]::new;
Function<String, StringBuilder> toBuf = StringBuilder::new;
```

Streams

Мотивирующий пример:

```
List<String> names = Arrays.asList(“Иван”, “Джон”, “”, “Хуан”);
// Нужно составить строку с приветствием непустых имен:
StringBuilder buf = new StringBuilder();
for (String name: names) {
    if (!name.isEmpty()) {
        if (buf.length() > 0) buf.append(“ “);
        buf.append(String.format(“Привет, %s!”, name));
    }
}
String greetings = buf.toString();
```

Streams

Мотивирующий пример:

```
List<String> names = Arrays.asList(“Иван”, “Джон”, “”, “Хуан”);
// Нужно составить строку с приветствием непустых имен:
String greetings = names.stream()
    .filter(name -> !name.isEmpty())
    .map(name -> String.format(“Привет, %s!”, name));
    .collect(joining(“ “));
```

Код выглядит более компактно и читаемо.

Такой подход позаимствован из функциональных языков.

Streams

В приведенном примере предполагается, что используется директива статического импорта:

```
import static java.util.Collectors.joining;
```

Директивы статического импорта позволяют использовать статические члены класса без указания имени класса, т.е. просто joining(“ “) вместо Collectors.joining(“ “).

Streams

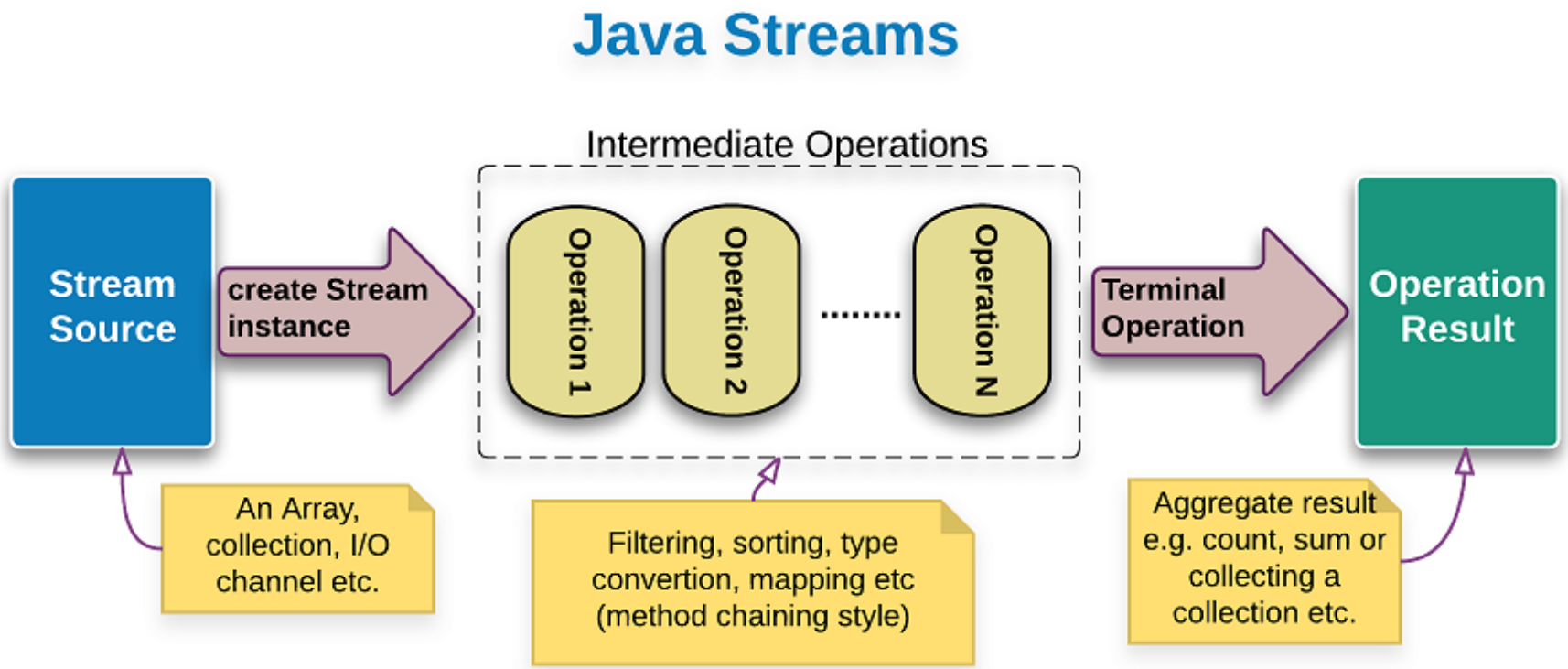
В интерфейсе Collection объявлен метод stream(), возвращающий поток элементов коллекции.

Поток (Stream) – это представление последовательности элементов, над которым можно производить операции. Операции делятся на две категории:

1. Нетерминальные – их результат тоже является потоком, и к нему в свою очередь можно тоже применить операцию;
2. Терминальные – их результат уже не является потоком, и эти операции завершают работу с потоком:

```
collection.stream().nonTerm1().nonTerm2()....nonTermN().term();
```

Streams



LogicBig.com

Streams

Нетерминальные операции потока Stream<T>:

1. Stream<T> filter(Predicate<T> predicate)
2. Stream<R> map(Function<T, R> mapper)
3. Stream<T> sorted()
4. Stream<T> distinct()
5. Stream<T> limit(long maxSize)
6. Stream<T> skip(long n)
7. Stream<R> flatMap(Function<T, Stream<R>> mapper)

(приведены чуть упрощенные сигнатуры методов, см. https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/stream/Stream.html)

Streams

Каждая нетерминальная операция преобразовывает последовательность элементов, и возвращает поток, соответствующий новой последовательности. Например, метод sorted() сортирует последовательность элементов и возвращает поток уже отсортированных элементов.

Метод skip(n) возвращает последовательность без n первых элементов.

Самые часто используемые методы – filter и map. Метод filter оставляет в последовательности только те элементы, которые удовлетворяют условию. Метод map применяет функцию к каждому элементу последовательности, получая новую последовательность.

Streams

Важная особенность нетерминальных методов – они являются ленивыми, т.е. при если у нас есть список из 1 миллиона элементов:

List<String> million = ...

то вызов

million.stream().map(str -> str + “!”)

не создает в памяти новый список из миллиона строк, а создает всего лишь новый объект Stream, который знает, что элементы нужно преобразовывать по мере необходимости.

Streams

Терминальные операции потока Stream<T>:

1. R collect(Collector<T, A, R> collector)
2. boolean anyMatch(Predicate<T> predicate)
3. long count()
4. Optional<T> findAny()
5. Optional<T> findFirst()
6. Optional<T> min(Comparator<T> comparator)
7. Optional<T> max(Comparator<T> comparator)
8. T reduce(T identity, BinaryOperator<T> op)
9. void forEach(Consumer<T> action)
10. ...

Streams

Терминальная операция “потребляет” поток, и любые другие вызовы операций на “истраченном” потоке вызовут ошибку.

Например, метод findFirst возвращает первый элемент потока (если он есть). При этом остальные элементы потока не будут обработаны вообще:

```
Optional<String> first = Stream.of(“A”, “B”, “C”).map(str -> {
    System.out.printf(“Processing %s\n”, str);
    return str;
}).findFirst();
```

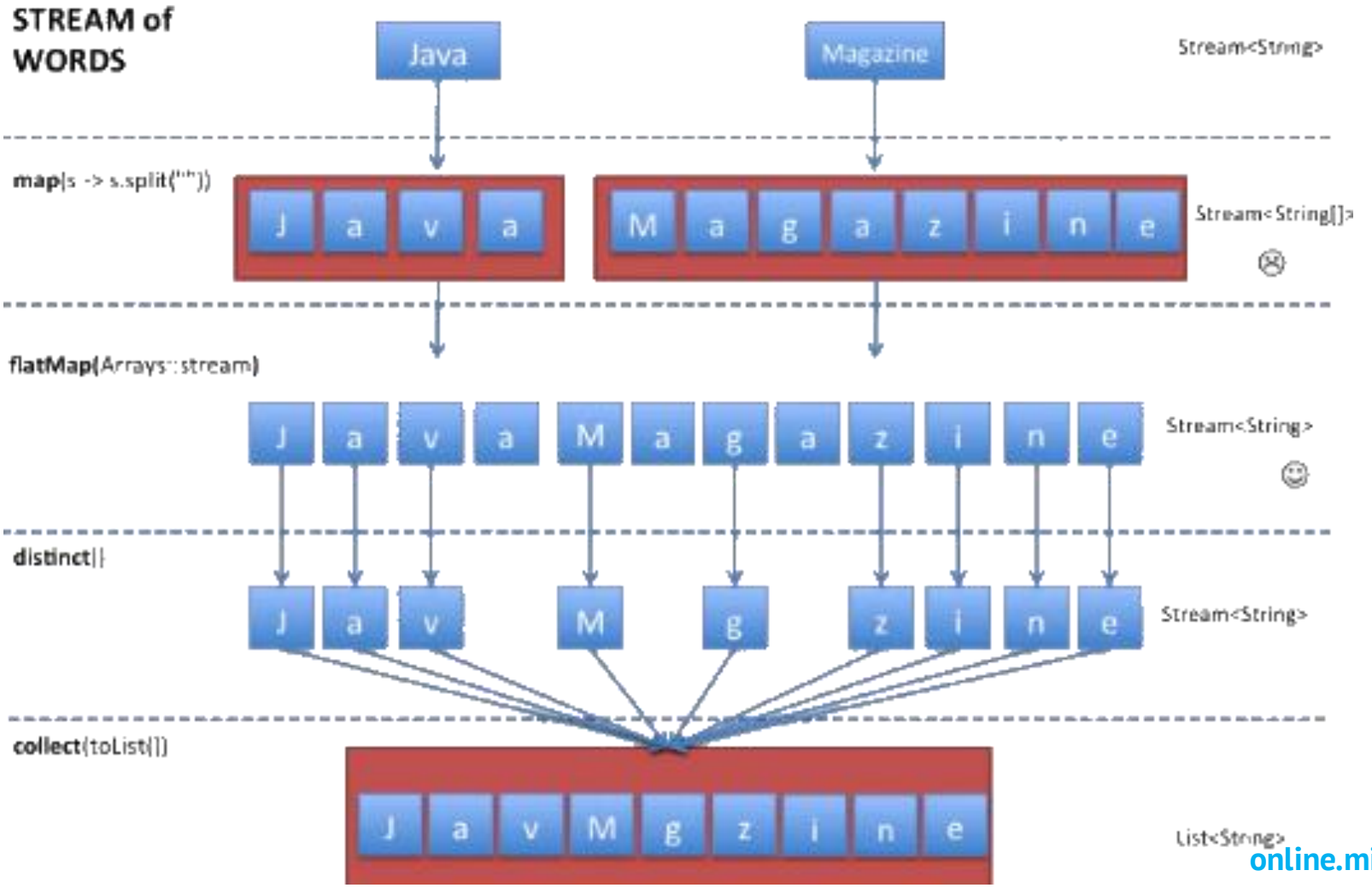
Будет напечатано только “Processing A”.

Streams

Таким образом, благодаря ленивости потоков, мы сэкономили ресурсы процессора, не применяя функцию, переданную в map, ко всем элементам последовательности, а только к тем, каким необходимо для получения результата терминальной операции.

Метод findAny отличается от метода findFirst тем, что он может быть более эффективен при параллельной обработке элементов, если нам не обязательно нужен первый элемент последовательности.

Streams


STREAM of WORDS
Java
Magazine
Stream<String>
map(s -> s.split(""))
J a v a
M a g a z i n e
Stream<String[]>
flatMap(Arrays::stream)
J a v a M a g a z i n e
Stream<String>
distinct()
J a v M g z i n e
Stream<String>
collect(toList())
J a v M g z i n e
List<String>

Streams

Не в любом случае стоит заменять цикл for на stream.

Например, когда вы на каждой итерации цикла используете предыдущий или следующий элемент, то такую логику трудно выразить с помощью потоков.

Также слишком сложные (особенно вложенные) потоки могут быть сложнее для понимания, чем вложенные циклы.

Streams

Один из самых часто используемых терминальных методов – collect. Он ожидает параметр Collector:

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
}
```

Streams

Для него есть много готовых реализаций в классе Collectors:

1. List<T> toList()
2. Set<T> toSet()
3. String joining(String separator)
4. Map<K, List<T>> groupingBy(Function<T, K> key)
5. ...

Streams

Пример вызова:

```
List<String> list = ...;
List<Integer> lengths = list.stream()
    .map(String::length)
    .collect(Collectors.toList());
```

Streams

В интерфейсе Collector<T, A, R> параметры:

1. T – тип элементов потока
2. A – тип, используемый внутри Collector для накопления результатов
3. R – тип финального результата метода collect

Пример – суммирующий коллектор (см. далее):

```
List<Integer> list = ...;
long sum = list.stream().collect(SumCollector.INSTANCE);
```

Streams

```
public class SumAcc { long sum = 0; }

public class SumCollector implements Collector<Number, SumAcc, Long> {
    public static final SumCollector INSTANCE = new SumCollector();
    public Supplier<SumAcc> supplier() {
        return () -> new SumAcc(); // Создание новой суммы
    }
    public BiConsumer<SumAcc, Number> accumulator() {
        return (acc, elem) -> acc.sum += elem.longValue(); // Добавление элемента в сумму
    }
    public BinaryOperator<SumAcc> combiner() {
        return (acc1, acc2) -> { // Композиция двух сумм
            acc1.sum += acc2.sum;
            return acc1;
        };
    }
    public Function<SumAcc, Long> finisher() {
        return acc -> acc.sum; // Извлечение финальной суммы
    }
}
```

Ссылка на код

Streams

На самом деле посчитать сумму можно и с использованием стандартных классов:

1. **long** sum = list.stream()
   .mapToLong(i -> i.longValue()) // LongStream
   .sum();
2. **long** sum = list.stream()
   .collect(summingLong(i -> i.longValue()));

Streams

Потоки своими руками.

Можно довольно легко написать упрощенную версию потоков на основе итераторов.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
}
```

Streams

```
// Обертка, преобразующая элементы исходного итератора:
public class MappedIterator<F, T> implements Iterator<T> {

   private final Iterator<F> original;
   private final Function<F, T> function;

   public MappedIterator(Iterator<F> original, Function<F, T> function) {
      this.original = original;
      this.function = function;
   }

   public boolean hasNext() {
      return original.hasNext();
   }

   public T next() {
      F elem = original.next();
      return function.apply(elem);
   }
}
```

Ссылка на код

Streams

```
// Обертка, фильтрующая элементы исходного итератора:
public class FilteredIterator<E> implements Iterator<E> {

   private final Iterator<E> original;
   private final Predicate<E> predicate;

   private boolean nextExists = false;
   private E next = null;

   public FilteredIterator(Iterator<E> original, Predicate<E> predicate) {
      this.original = original;
      this.predicate = predicate;
   }
```

Ссылка на код

Streams

```
public boolean hasNext() {
   if (nextExists)
      return true;
   while (original.hasNext()) {
      E elem = original.next();
      if (predicate.test(elem)) {
         nextExists = true;
         next = elem;
         return true;
      }
   }
   return false;
}
```

```
public E next() {
   if (hasNext()) {
      E result = next;
      nextExists = false;
      next = null;
      return result;
   } else {
      throw new NoSuchElementException();
   }
}
```

Streams

```
// Аналог Stream на основе Iterator с map и filter:
public class MyStream<E> {

    private final Iterator<E> iterator;

    public MyStream(Iterator<E> iterator) {
        this.iterator = iterator;
    }

    public <T> MyStream<T> map(Function<E, T> function) {
        return new MyStream<>(new MappedIterator<>(iterator, function));
    }

    public MyStream<E> filter(Predicate<E> predicate) {
        return new MyStream<>(new FilteredIterator<>(iterator, predicate));
    }
```

Ссылка на код

Streams

```
public List<E> toList() {
    List<E> list = new ArrayList<>();
    while (iterator.hasNext()) {
        E elem = iterator.next();
        list.add(elem);
    }
    return list;
}

public static void main(String[] args) {
    List<String> list = Arrays.asList("Иван", "Джон", "", "Хуан");
    List<String> greetings = new MyStream<>(list.iterator())
        .filter(str -> !str.isEmpty())
        .map(name -> String.format("Привет, %s!", name))
        .toList();
    System.out.println(greetings);
}
```

Streams

Реальный класс Stream работает похоже, но использует не Iterator, а Spliterator:

```
public interface Spliterator<T> {
    boolean tryAdvance(Consumer<T> action);
    Spliterator<T> trySplit();
    long estimateSize();
}
```

Streams

Методы Spliterator:

1. tryAdvance: полный аналог методов hasNext+next из Iterator; объединены в один, так как так его проще реализовывать (см. пример)
2. Метод estimateSize: может использоваться для оптимизации; например, в нашем MyStream мы могли бы его использовать для начального размера списка в методе toList
3. Метод trySplit: используется для возможности **параллельного обхода коллекции**

Streams

Довольно значительная часть особенностей Stream API связана с тем, что в них реализована возможность параллельной обработки элементов в нескольких потоках (threads). Так можно ускорить обработку данных:

```
List<String> list = Arrays.asList("A", "B");
int sumLen = list.stream().parallel().mapToInt(str -> {
    Thread.sleep(1000); return str.length();
}).sum();
// Выполняется 1 секунду, а не 2: см. код
```

Streams

При параллельной обработке stream вызывает метод Spliterator.trySplit, который делит последовательность элементов на две примерно равных под-последовательности (если это невозможно, например если в последовательности только один элемент, trySplit возвращает null). Затем каждую под-последовательность Stream обрабатывает в своем собственном thread. Это разбиение может продолжаться и далее, пока не будет достигнуто оптимальное количество threads.

Streams

```
private static int sum(Spliterator.OfInt s) {
    int[] sum = {0};
    s.forEachRemaining((int i) -> sum[0] += i);
    return sum[0];
}
// Пример использования Spliterator для параллелизации вычислений:
public static int parallelSum(Spliterator.OfInt si) {
    Spliterator.OfInt split = si.trySplit();
    if (split != null) {
        ExecutorService pool = ForkJoinPool.commonPool();
        Future<Integer> f1 = pool.submit(() -> sum(si));
        Future<Integer> f2 = pool.submit(() -> sum(split));
        return f1.get().intValue() + f2.get().intValue();
    } else {
        return sum(si);
    }
}
```

Ссылка на код

Streams

```
public static void main(String[] args) throws Exception {
    int[] array = new int[500_000_000];
    Random rnd = new Random(0);
    for (int j = 0; j < array.length; j++) {
        array[j] = rnd.nextInt();
    }
    System.out.println("START");
    Spliterator.OfInt arraySplit = Arrays.spliterator(array);
    long t1 = System.currentTimeMillis();
    int sum = parallelSum(arraySplit);
    long t2 = System.currentTimeMillis();
    System.out.println(sum);
    System.out.println(t2 - t1);
}
```

sum: 400 мс
parallelSum: 250 мс

Streams

Интерфейс Spliterator редко используется напрямую в программах, хотя и является основой для Stream.

Из Spliterator можно создать Stream:

```
Stream<T> StreamSupport.stream(
    Spliterator<T> spliterator,
    boolean parallel)
```

И наоборот, из Stream получить Spliterator можно через метод Stream.spliterator().

Streams

Для корректности параллельной работы потоков нужно, чтобы функции, передаваемые в методы типа map или filter были чистыми, т.е. использовали только свои параметры и, возможно, неизменяемые объекты (т.к. неизменяемые объекты можно безопасно использовать в нескольких потоках).

```
int sum = 0; // поле класса

...

list.stream().parallel().forEach(n -> sum += n);
```

Здесь функция имеет побочный эффект, что может вызвать проблемы при параллельной обработке.