СБЕРБАНК ТЕХНОЛОГИИ

Lambda, Stream API, JAVA 8 features

УЗНАЕМ

СБЕРБАНК ТЕХНОЛОГИИ

- **Что нового в java 8?**
- **Что-такое лямбда, зачем она нужна, когда использовать?**
- **Что-такое Stream API, какие задачи оно решает?**
- **Зачем нужны дефолтные методы интерфейсу?**

ЧТО НОВОГО

СБЕРБАНК ТЕХНОЛОГИИ

1. Введён функциональный стиль программирования
2. Лябмда выражения
3. Ссылки на методы
4. Дефолтные методы интерфейса
5. Новый подход работы с датой и временем (Date-Time Package)
6. JavaScript Engine – Nashorn
7. Optional класс для уменьшения NullPointerException
8. CompletableFuture для компануемого асинхронного программирования
9. Повторяющиеся аннотации
10. Улучшения в коллекциях

ПОСТАНОВКА ЗАДАЧИ

СБЕРБАНК ТЕХНОЛОГИИ

Необходимо написать функцию сортировки зелёных яблок по различным критериям.

```
public static List<Apple> filterApples(List<Apple> inventory, ????);
```

BEHAVIOR PARAMETERIZATION

СБЕРБАНК ТЕХНОЛОГИИ

behavior parameterization - это шаблон проектирования, позволяющий быстро реагировать на изменения требований.

BEHAVIOR PARAMETERIZATION

СБЕРБАНК ТЕХНОЛОГИИ

behavior parameterization - это шаблон проектирования, позволяющий быстро реагировать на изменения требований.

Технически, behavior parameterization – это возможность сказать методу взять множество стратегий как параметры и использовать их внутри себя.

BEHAVIOR PARAMETERIZATION

СБЕРБАНК ТЕХНОЛОГИИ

behavior parameterization - это шаблон проектирования, позволяющий быстро реагировать на изменения требований.

Технически, behavior parameterization – это возможность сказать методу взять множество стратегий как параметры и использовать их внутри себя.

По сути позволяет создавать ссылку на блок кода, без его выполнения.

Этот участок кода может быть выполнен потом в другом месте программы.

Например, можно передать участок кода как аргумент для функции.

BEHAVIOR PARAMETERIZATION

СБЕРБАНК ТЕХНОЛОГИИ

behavior parameterization - это шаблон проектирования, позволяющий быстро реагировать на изменения требований.

Технически, behavior parameterization – это возможность сказать методу взять множество стратегий как параметры и использовать их внутри себя.

По сути позволяет создавать ссылку на блок кода, без его выполнения.

Этот участок кода может быть выполнен потом в другом месте программы.

Например, можно передать участок кода как аргумент для функции.

На выходе получаем очень гибкий код.

РЕШЕНИЕ ДО JAVA 8

СБЕРБАНК ТЕХНОЛОГИИ

Определим интерфейс предиката:

```
public interface ApplePredicate{
    boolean test (Apple apple);
}
```

РЕШЕНИЕ ДО JAVA 8

СБЕРБАНК ТЕХНОЛОГИИ

Определим интерфейс предиката:

```
public interface ApplePredicate{
    boolean test (Apple apple);
}
```

Реализуем саму функцию:

```
public static List<Apple> filterApples(List<Apple> inventory, ApplePredicate p) {
    List<Apple> result = new ArrayList<>();
    for(Apple a : inventory) {
        if(p.test(a)) {
            result.add(a);
        }
    }
    return result;
}
```

РЕШЕНИЕ ДО JAVA 8 - ИСПОЛЬЗОВАНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Определим классы стратегий:

```
public class AppleGreenColorPredicate implements ApplePredicate {
    @Override
    public boolean test(Apple apple) {
        return "green".equals(apple.getColor());
    }
}
public class AppleHeavyWeightPredicate implements ApplePredicate {
    @Override
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}
```

РЕШЕНИЕ ДО JAVA 8 - ИСПОЛЬЗОВАНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Определим классы стратегий:

```
public class AppleGreenColorPredicate implements ApplePredicate {
    @Override
    public boolean test(Apple apple) {
        return "green".equals(apple.getColor());
    }
}
public class AppleHeavyWeightPredicate implements ApplePredicate {
    @Override
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}
```

Вызовы методов:

```
List<Apple> redAndHeavyApples =
            filterApples(inventory, new AppleHeavyWeightPredicate());
List<Apple> redAndHeavyApples =
            filterApples(inventory, new AppleGreenColorPredicatePredicate());
```

РЕШЕНИЕ ДО JAVA 8 - ИСПОЛЬЗОВАНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Определим классы стратегий:

```
public class AppleGreenColorPredicate implements ApplePredicate {
    @Override
    public boolean test(Apple apple) {
        return "green".equals(apple.getColor());
    }
}
public class AppleHeavyWeightPredicate implements ApplePredicate {
    @Override
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}
```

Вызовы методов:

```
List<Apple> redAndHeavyApples =
            filterApples(inventory, new AppleHeavyWeightPredicate());
List<Apple> redAndHeavyApples =
            filterApples(inventory, new AppleGreenColorPredicatePredicate());
```

Слишком много обслуживающего кода

РЕШЕНИЕ ДО JAVA 8 – АНОНИМНЫЕ КЛАССЫ

СБЕРБАНК ТЕХНОЛОГИИ

Можно короче – через анонимные классы:

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {
    @Override
    public boolean test(Apple apple) {
        return "green".equals(apple.getColor());
    }
});

List<Apple> heavyApples = filterApples(inventory, new ApplePredicate() {
    @Override
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
});
```

РЕШЕНИЕ ДО JAVA 8 – АНОНИМНЫЕ КЛАССЫ

СБЕРБАНК ТЕХНОЛОГИИ

Можно короче – через анонимные классы:

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {
    @Override
    public boolean test(Apple apple) {
        return "green".equals(apple.getColor());
    }
});

List<Apple> heavyApples = filterApples(inventory, new ApplePredicate() {
    @Override
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
});
```

Уже лучше – более компактно

РЕШЕНИЕ JAVA 8

СБЕРБАНК ТЕХНОЛОГИИ

```
List<Apple> redApples = filterApples(inventory, (Apple a) -> "green".equals(a.getColor()));

List<Apple> heavyApples = filterApples(inventory, (Apple a) -> a.getWeight() > 150);
```

РЕШЕНИЕ JAVA 8

СБЕРБАНК ТЕХНОЛОГИИ

```
List<Apple> redApples = filterApples(inventory, (Apple a) -> "green".equals(a.getColor()));

List<Apple> heavyApples = filterApples(inventory, (Apple a) -> a.getWeight() > 150);
```

Гораздо меньше кода. Только по существу.

LAMBDA ВЫРАЖЕНИЯ

СБЕРБАНК ТЕХНОЛОГИИ

К lambda функции можно относиться как к анонимному методу, без имени, который может быть передан как аргумент, так же как объект анонимного класса.

КАК ВЫГЛЯДИТ

- Список параметров, в некоторых случаях тип можно не задавать
- Стрелка разделяет список параметров от тела
- Само тело, в некоторых случаях должно быть в скобках

ПРИМЕРЫ ВАЛИДНЫХ LAMBDA ВЫРАЖЕНИЙ

СБЕРБАНК ТЕХНОЛОГИИ

Случай использования	Пример lambda
Логическое выражение	`(List<String> list) -> list.isEmpty();`
Создание объекта	`() -> new Apple(10);`
Получение из объекта	`(Apple a) -> {System.out.println(a.getWeight());}`
Объединение двух значение	`(int a, int b) -> a * b;`
Сравнение двух объектов	`(Apple a1, Apple a2) ->` `  a1.getWeight().compareTo(a2.getWeight());`

ОСНОВНЫЕ СВОЙСТВА LAMBDA

СБЕРБАНК ТЕХНОЛОГИИ

- Анонимность – не имеет имени
- Функция – потому-что не относится к какому-либо классу, как метод, но имеет параметры и т.д.
- ***Passed around –*** может быть передана как аргумент в метод или сохранена в переменную
- Лаконичность – не требует написание обслуживающего кода, как для анонимного класса

ГДЕ И КОГДА МОЖНО ИСПОЛЬЗОВАТЬ LAMBDA

СБЕРБАНК ТЕХНОЛОГИИ

Вспомним код:

```
List<Apple> redApples = filter(inventory, (Apple apple) -> "red".equals(apple.getColor()));
```

ГДЕ И КОГДА МОЖНО ИСПОЛЬЗОВАТЬ LAMBDA

СБЕРБАНК ТЕХНОЛОГИИ

Вспомним код:

```
List<Apple> redApples = filter(inventory, (Apple apple) -> "red".equals(apple.getColor()));
```

Мы смогли передали lambda, т.к. метод принимает интерфейс Predicate<T>

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

ГДЕ И КОГДА МОЖНО ИСПОЛЬЗОВАТЬ LAMBDA

СБЕРБАНК ТЕХНОЛОГИИ

- Такой интерфейс с одним абстрактным методом называется функциональным.
- Только если параметр метода имеет тип функционального интерфейса, ему можно передать lambda выражение.
- Аннотация @FunctionalInterface помогает компилятору производить проверку интерфейса на соответствие признакам функционального.

ОСНОВНЫЕ ФУНКЦИОНАЛЬНЫЕ ИНТЕРФЕЙСЫ JDK8

СБЕРБАНК ТЕХНОЛОГИИ

Интерфейс	Назначение

`@FunctionalInterface`
`public interface Predicate<T>{`
`  boolean test(T t);`
`}`
	Проверяет на соответствие условию

`@FunctionalInterface`
`public interface Consumer<T> {`
`  void accept(T t);`
`}`
	Получает объект.

`@FunctionalInterface`
`public interface Function<T, R> {`
`  R apply(T t);`
`}`
	Получает объект, возвращает результат.

`@FunctionalInterface`
`public interface Supplier<T> {`
`  T get();`
`}`
	Поставщик

FUNCTION DESCRIPTOR

СБЕРБАНК ТЕХНОЛОГИИ

Сигнатура абстрактного метода в функциональном интерфейсе отражает сигнатуру lambda выражения и называется *function description.*

FUNCTION DESCRIPTOR ПРИМЕРЫ ИЗ JDK

СБЕРБАНК ТЕХНОЛОГИИ

Функциональный интерфейс	Function descriptor

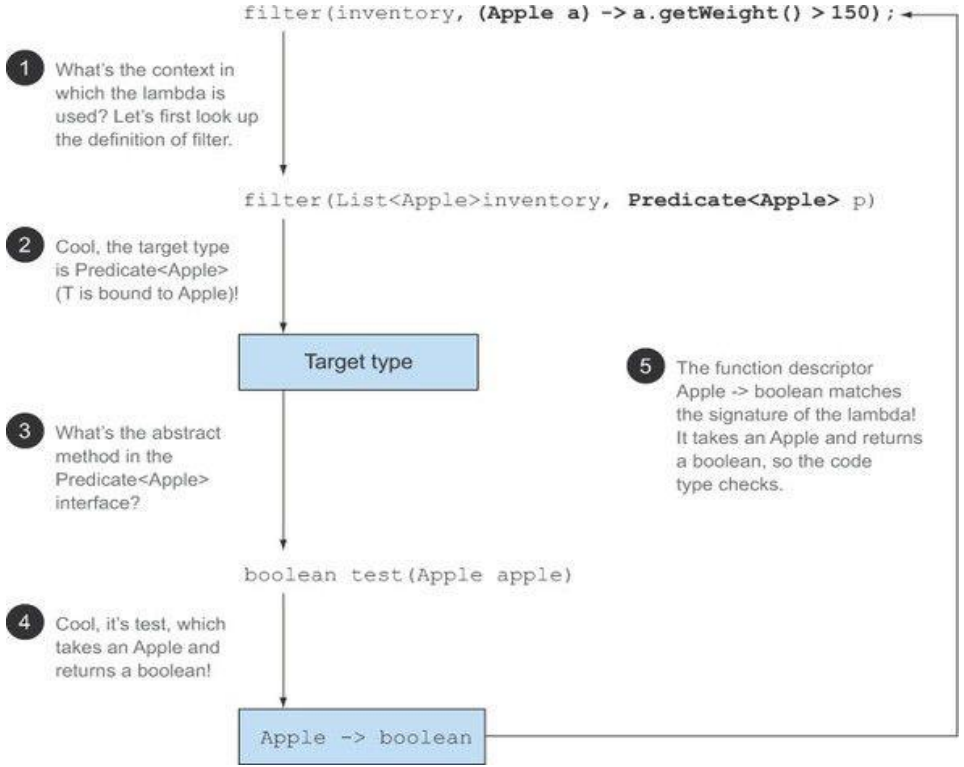
`@FunctionalInterface`
`public interface Predicate<T> {`
`  boolean test(T t);`
`}`
	T -> boolean

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

ПРОВЕРКА ТИПОВ

СБЕРБАНК ТЕХНОЛОГИИ

Тип лямбда выражения выводится из контекста его использования следующим образом:



ВЫВЕДЕНИЕ ТИПОВ

СБЕРБАНК ТЕХНОЛОГИИ

Так как компилятор выводит из контекста использования лямбда функциональный интерефейс, то он знает и function descriptor.

ВЫВЕДЕНИЕ ТИПОВ

СБЕРБАНК ТЕХНОЛОГИИ

Следовательно можно опустить типы параметров:

```
List<Apple> redApples =
        filter(inventory, (Apple apple) -> "red".equals(apple.getColor()));

List<Apple> redApples =
        filter(inventory, apple -> "red".equals(apple.getColor()));
```

ВЫВЕДЕНИЕ ТИПОВ

СБЕРБАНК ТЕХНОЛОГИИ

Следовательно можно опустить типы параметров:

```
List<Apple> redApples =
        filter(inventory, (Apple apple) -> "red".equals(apple.getColor()));

List<Apple> redApples =
        filter(inventory, apple -> "red".equals(apple.getColor()));

List<Apple> redApples =
        filter(inventory, (Apple a1, Apple a2) -> a1.getWeight().
          compareTo(a2.getWeight()));

List<Apple> redApples =
        filter(inventory, (a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

ЛЯМБДА ВЫРАЖЕНИЯ МОЖНО СОЧЕТАТЬ

СБЕРБАНК ТЕХНОЛОГИИ

Пример объединения компараторов

```
inventory.sort(
        Comparator.comparing(Apple::getWeight)
        .reversed()
        .thenComparing(Apple::getCountry)
);
```

КАК ПРОКИНУТЬ ИСКЛЮЧЕНИЕ ИЗ LAMBDA

СБЕРБАНК ТЕХНОЛОГИИ

1. Описываем функциональный интерфейс

```
@FunctionalInterface
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}
BufferedReaderProcessor p = (BufferedReader br) -> br.readLine();
```

КАК ПРОКИНУТЬ ИСКЛЮЧЕНИЕ ИЗ LAMBDA

СБЕРБАНК ТЕХНОЛОГИИ

2. Использовать стандартные и оборачивать в Runtime исключения

```
Function<BufferedReader, String> f = (BufferedReader b) -> {
            try {
                return b.readLine();
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        };
```

ССЫЛКИ НА МЕТОДЫ

СБЕРБАНК ТЕХНОЛОГИИ

Ссылки на метод позволяют переиспользовать существующий метод, передавая его как лямбда.

```
inventory.sort((Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));

inventory.sort(Comparator.comparing(Apple::getWeight));
```

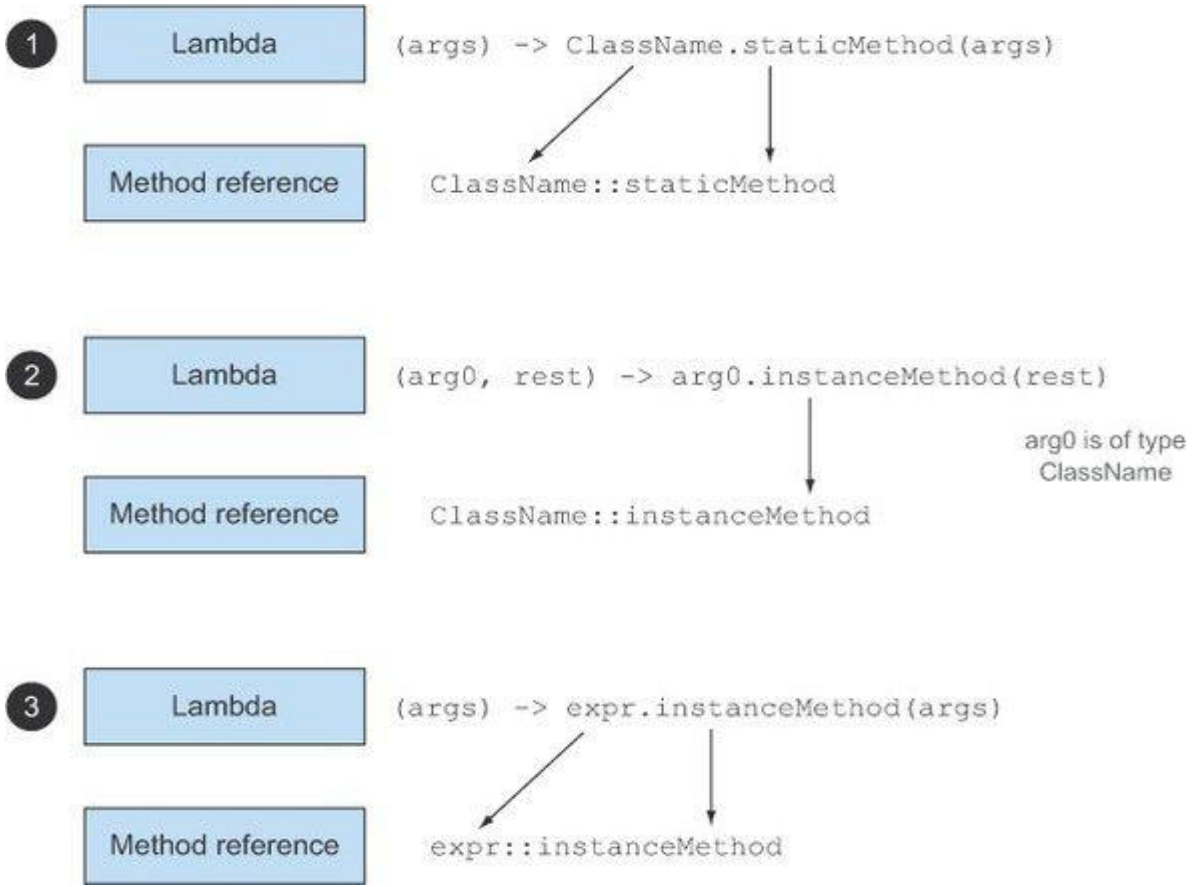
ОСНОВНЫЕ ТИПЫ ССЫЛОК НА МЕТОДЫ

СБЕРБАНК ТЕХНОЛОГИИ

- Ссылка на статический метод
- Ссылка на обычный метод произвольного типа
- Ссылка на метод конкретного объекта
- Ссылка на конструктор

ПРИМЕРЫ

СБЕРБАНК ТЕХНОЛОГИИ



LAMBDA ВЫРАЖЕНИЯ ИТОГИ

СБЕРБАНК ТЕХНОЛОГИИ

- Лябмда выражение – анонимная функция, не имеет имени, но имеет параметры, тело, возвращаемый тип, и список исключений
- Лябмда выражение позволяет сокращать код, избавляет от обслуживающего кода
- Лябмда выражение может быть использовано только там где ожидается функциональный интерфейс
- Лябмда выражение позволяет реализовывать абстрактный метод функционального интерфейса в точке его передачи в метод и относиться к нему как к объекту функционального интерфейса
- В JAVA 8 существует ряд встроенных функциональных интерфейсов (java.util.function).
- Ссылки на методы можно использовать в тех местах где ожидается функциональный интерефейс

ПОЧЕМУ ПОЯВИЛСЯ STREAM API

СБЕРБАНК ТЕХНОЛОГИИ

Работа с коллекциями не всегда идеальная.

Иногда приходится писать много «обслуживающего» циклы кода.

ПОЧЕМУ ПОЯВИЛСЯ STREAM API

СБЕРБАНК ТЕХНОЛОГИИ

До java 8:

```
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish d : menu) {
    if(d.getCalories() < 400) {
        lowCaloricDishes.add(d);
    }
}
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
    @Override
    public int compare(Dish o1, Dish o2) {
        return Integer.compare(o1.getCalories(), o2.getCalories());
    }
});
List<String> lowCaloricDishesName = new ArrayList<>();
for(Dish d : lowCaloricDishes) {
    lowCaloricDishesName.add(d.getName());
}
```

ПОЧЕМУ ПОЯВИЛСЯ STREAM API

СБЕРБАНК ТЕХНОЛОГИИ

После java 8:

```
List<String> lowCaloricDishesName = menu.stream()
        .filter(d -> d.getCalories() < 400)
        .sorted(Comparator.comparing(Dish::getCalories))
        .map(Dish::getName)
        .collect(toList());
```

ПОЧЕМУ ПОЯВИЛСЯ STREAM API

СБЕРБАНК ТЕХНОЛОГИИ

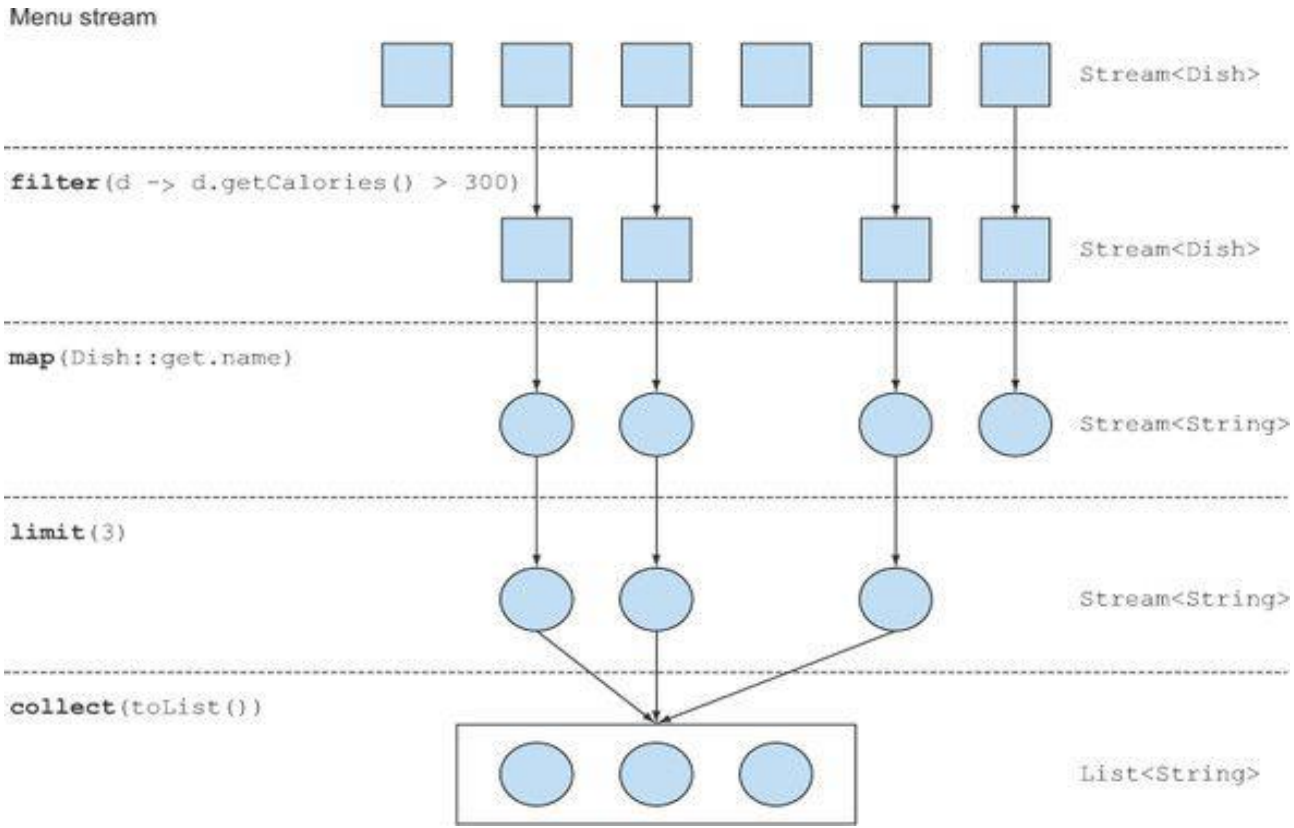
После java 8:

```
List<String> lowCaloricDishesName = menu.stream()
        .filter(d -> d.getCalories() < 400)
        .sorted(Comparator.comparing(Dish::getCalories))
        .map(Dish::getName)
        .collect(toList());
```

- Код написан в декларативной манере – написано что нужно достичь, а не то как это надо реализовать.
- Поддержка pipeline позволяет писать сложные цепочки обработки данных
- В java 8 — внутреннее итерирование — циклы переносятся в реализацию библиотек

STREAM ПРИМЕР ВИЗУАЛИЗАЦИЯ

СБЕРБАНК ТЕХНОЛОГИИ



ПОЧЕМУ ПОЯВИЛСЯ STREAM API

СБЕРБАНК ТЕХНОЛОГИИ

Что если у нас большой объём данных и несколько CPU?

Очень медленно обрабатывать последовательно.

Как задействовать все имеющиеся CPU на машине и обрабатывать параллельно?

ПОЧЕМУ ПОЯВИЛСЯ STREAM API

СБЕРБАНК ТЕХНОЛОГИИ

Что если у нас большой объём данных и несколько CPU?

Очень медленно обрабатывать последовательно.

Как задействовать все имеющиеся CPU на машине и обрабатывать параллельно?

Многопоточный подход.

ПРОБЛЕМЫ ПАРАЛЛЕЛЬНОЙ ОБРАБОТКИ

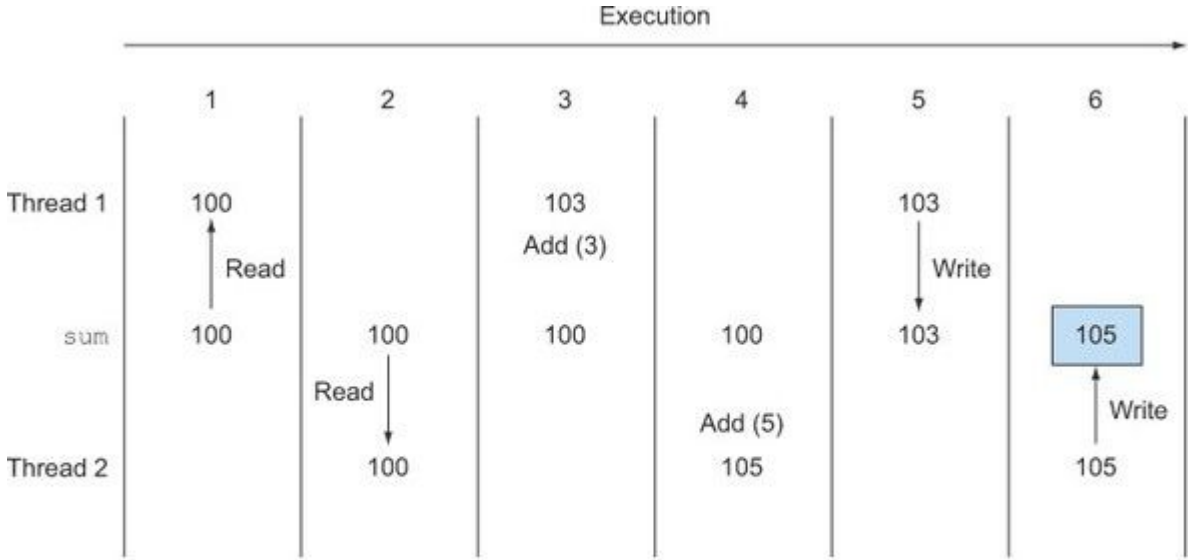
СБЕРБАНК ТЕХНОЛОГИИ

- писать многопоточный код — сложно и громоздко
- необходимо решать проблемы синхронизации

ПРОБЛЕМЫ ПАРАЛЛЕЛЬНОЙ ОБРАБОТКИ

СБЕРБАНК ТЕХНОЛОГИИ

- писать многопоточный код — сложно и громоздко
- необходимо решать проблемы синхронизации



STREAM И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА

СБЕРБАНК ТЕХНОЛОГИИ

Параллелизм в java 8 stream api практически бесплатный

STREAM И ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА

СБЕРБАНК ТЕХНОЛОГИИ

Параллелизм в java 8 stream api практически бесплатный

В большинстве случаев достаточно написать так:

```
List<Apple> heavyApples = inventory.parallelStream().filter((Apple a) ->
                a.getWeight() > 150).collect(toList());
```

STREAM ОПРЕДЕЛЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Stream – это последовательность элементов от источника, которые поддерживают процессинговые операции.

STREAM ОПРЕДЕЛЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Stream – это последовательность элементов от источника, которые поддерживают процессинговые операции.

- последовательность элементов – как коллекция, но не для хранения, а для процессинга
- источник – например коллекции, массивы, I/O ресурсы
- процессинговые операции – аналог SQL операций в базе данных и общих операций из функциональных языков, таких как filter, map, reduce, find, match, sort и т.п.

STREAM ВАЖНЕЙШИЕ СВОЙСТВА

СБЕРБАНК ТЕХНОЛОГИИ

- Объединение в цепочки обработки pipelines
- Внутреннее итерирование
- Обрабатывает данные только один раз

ОСНОВНЫЕ ИСТОЧНИКИ СТРИМОВ

СБЕРБАНК ТЕХНОЛОГИИ

- Интерфейс Collection имеет дефолтный метод stream(), следовательно можно получать stream из потомков (List, Set, Queue)

ОСНОВНЫЕ ИСТОЧНИКИ СТРИМОВ

СБЕРБАНК ТЕХНОЛОГИИ

- Интерфейс Collection имеет дефолтный метод stream(), следовательно можно получать stream из потомков (List, Set, Queue)
- Явное создание

```
Stream<String> stream = Stream.of("Java 8 ", "Lambdas ", "In ", "Action");
```

ОСНОВНЫЕ ИСТОЧНИКИ СТРИМОВ

СБЕРБАНК ТЕХНОЛОГИИ

- Интерфейс Collection имеет дефолтный метод stream(), следовательно можно получать stream из потомков (List, Set, Queue)
- Явное создание

```
Stream<String> stream = Stream.of("Java 8 ", "Lambdas ", "In ", "Action");
```

- Из массива элементов

```
Stream<Integer> strm = Arrays.stream(new Integer[]{1,2,3,4,5});
```

ОСНОВНЫЕ ИСТОЧНИКИ СТРИМОВ

СБЕРБАНК ТЕХНОЛОГИИ

- Интерфейс Collection имеет дефолтный метод stream(), следовательно можно получать stream из потомков (List, Set, Queue)
- Явное создание

```
Stream<String> stream = Stream.of("Java 8 ", "Lambdas ", "In ", "Action");
```

- Из массива элементов

```
Stream<Integer> strm = Arrays.stream(new Integer[]{1,2,3,4,5});
```

- Из файла

```
Stream<String> strm = Files.lines(new File("test.txt").toPath());
```

КЛАССИФИКАЦИЯ ОПЕРАЦИИ НАД STREAM

СБЕРБАНК ТЕХНОЛОГИИ

- Промежуточные (Intermediate)
- Терминальные (terminal)

КЛАССИФИКАЦИЯ ОПЕРАЦИИ НАД STREAM

СБЕРБАНК ТЕХНОЛОГИИ

- Промежуточные (Intermediate)
- Терминальные (terminal)

```
List<String> names = menu.stream()
        .filter(d -> d.getCalories() > 300)
        .map(Dish::getName)
        .limit(3)
        .collect(toList());
```

СВОЙСТВА INTERMEDIATE ОПЕРАЦИЙ

СБЕРБАНК ТЕХНОЛОГИИ

- Возвращают другой стрим
- Могут быть объединены в цепочку, аналог некой формы запроса
- Ленивые – не проводят никаких действий, пока на стриме не будет вызвана terminate операция

СВОЙСТВА TERMINATE ОПЕРАЦИЙ

СБЕРБАНК ТЕХНОЛОГИИ

- Формируют результат (не Stream типа: List, Integer, …) из pipeline стрима

ОСНОВНЫЕ INTERMEDIATE ОПЕРАЦИИ - FILTER

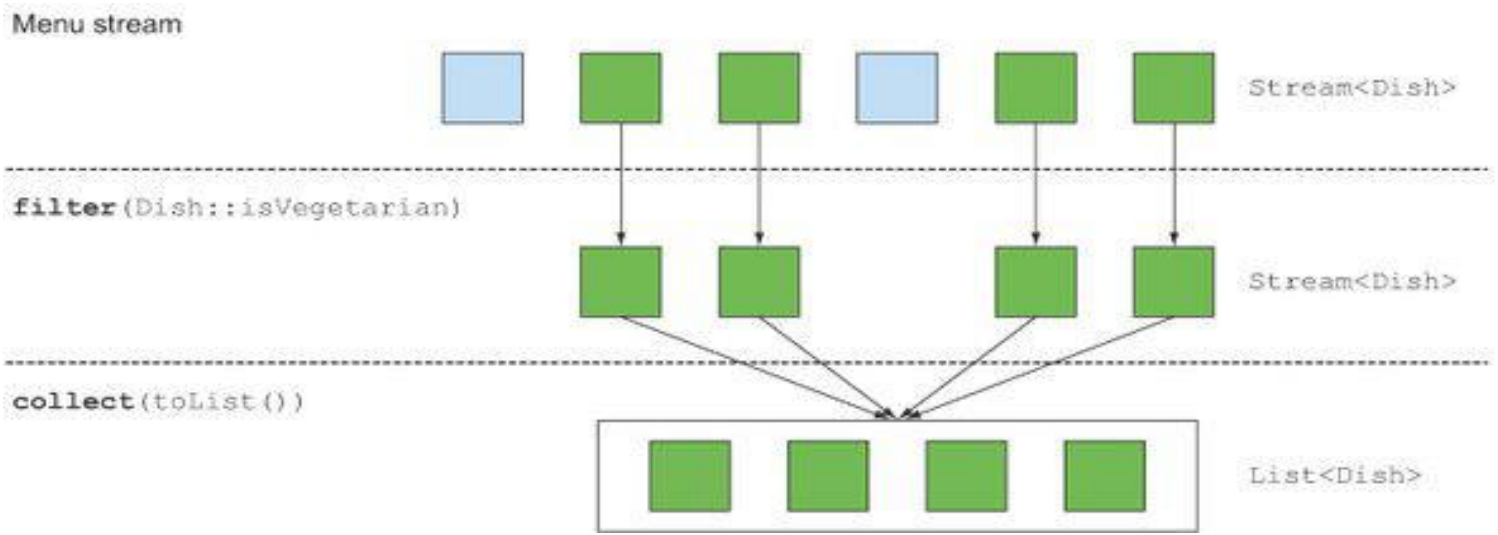
СБЕРБАНК ТЕХНОЛОГИИ

```
List<Dish> vegetarianMenu = menu.stream()
        .filter(Dish::isVegetarian)
        .collect(toList());
```

ОСНОВНЫЕ INTERMEDIATE ОПЕРАЦИИ - FILTER

СБЕРБАНК ТЕХНОЛОГИИ

```
List<Dish> vegetarianMenu = menu.stream()
        .filter(Dish::isVegetarian)
        .collect(toList());
```



ОСНОВНЫЕ INTERMEDIATE ОПЕРАЦИИ - FILTER

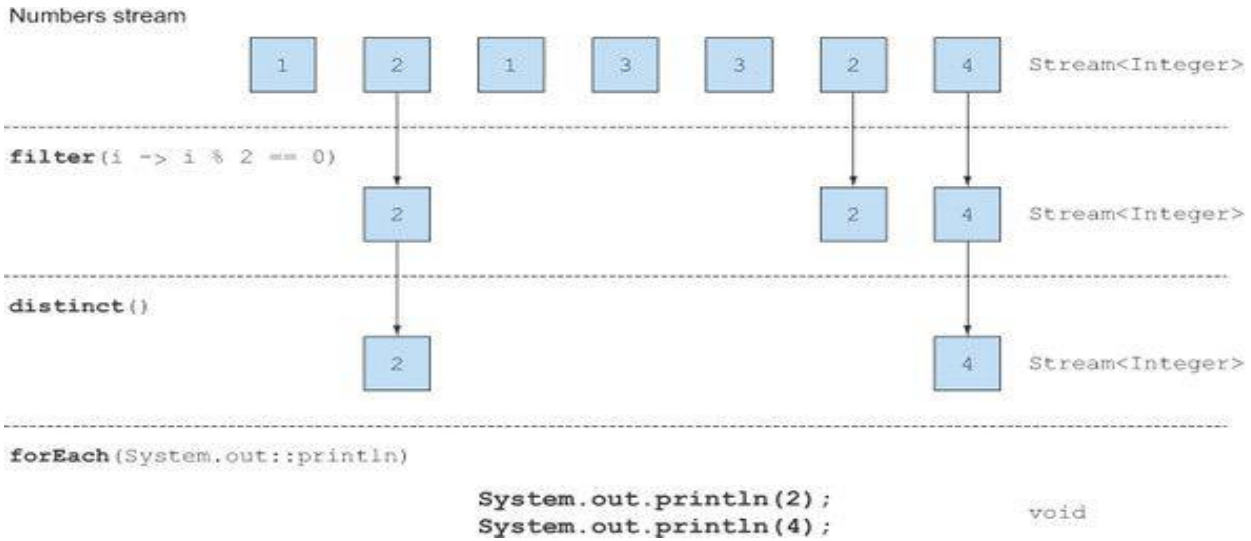
СБЕРБАНК ТЕХНОЛОГИИ

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
        .filter(i -> i % 2 == 0)
        .distinct()
        .forEach(System.out::println);
```

ОСНОВНЫЕ INTERMEDIATE ОПЕРАЦИИ - FILTER

СБЕРБАНК ТЕХНОЛОГИИ

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
        .filter(i -> i % 2 == 0)
        .distinct()
        .forEach(System.out::println);
```



ОСНОВНЫЕ INTERMEDIATE ОПЕРАЦИИ - MAP

СБЕРБАНК ТЕХНОЛОГИИ

```
List<String> dishNames = menu.stream()
        .map(Dish::getName)
        .collect(toList());
```

Преобразует стрим `Stream<Dish>` в `Stream<String>`

ОСНОВНЫЕ INTERMEDIATE ОПЕРАЦИИ - FLATMAP

СБЕРБАНК ТЕХНОЛОГИИ

Задача: разбить слова по буквам из входного потока:

["Hello", "World"] -> ["H", "e", "l", "o","W", "r", "d"]

ОСНОВНЫЕ INTERMEDIATE ОПЕРАЦИИ - FLATMAP

СБЕРБАНК ТЕХНОЛОГИИ

Задача: разбить слова по буквам из входного потока:

["Hello", "World"] -> ["H", "e", "l", "o","W", "r", "d"]

```
List<String> words = Arrays.asList("Hello world");
words.stream()
        .map(word -> word.split(""))
        .distinct()
        .collect(toList());
```

ОСНОВНЫЕ INTERMEDIATE ОПЕРАЦИИ - FLATMAP

СБЕРБАНК ТЕХНОЛОГИИ

Задача: разбить слова по буквам из входного потока:

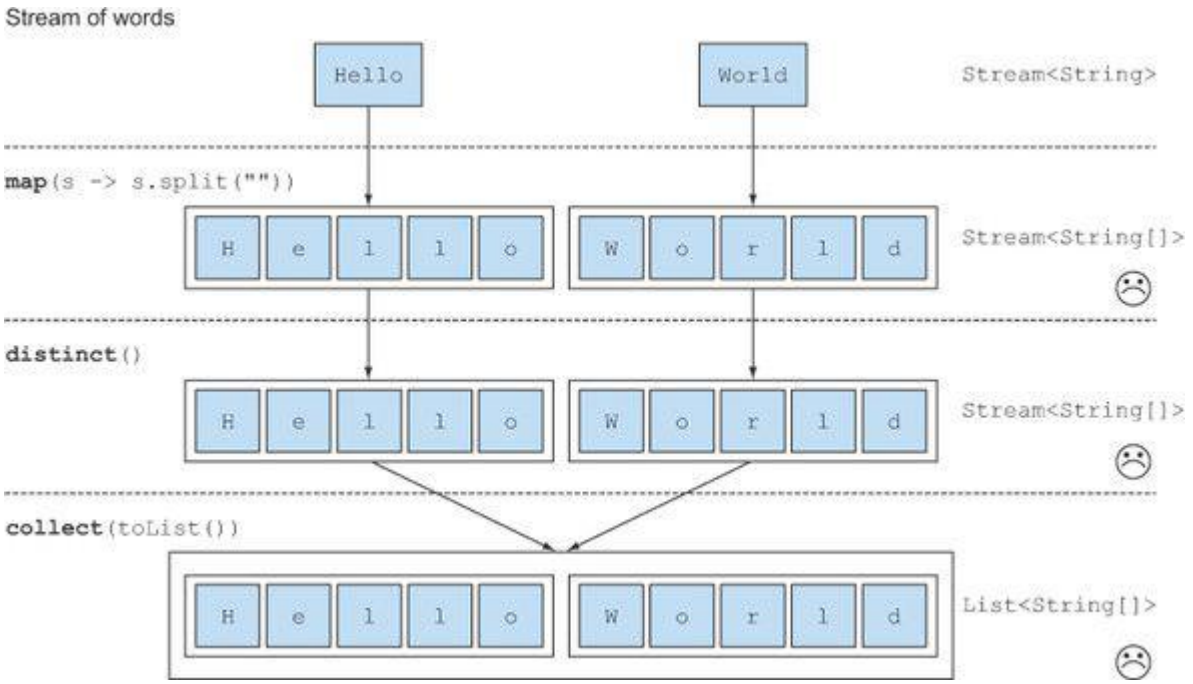
["Hello", "World"] -> ["H", "e", "l", "o","W", "r", "d"]

```
List<String> words = Arrays.asList("Hello world");
words.stream()
        .map(word -> word.split(""))
        .distinct()
        .collect(toList());
```

Проблема в том что map метод вернёт Stream<String[]>, а хотелось бы Stream<String>

FLATMAP - ПРОБЛЕМА

СБЕРБАНК ТЕХНОЛОГИИ



FLATMAP - РЕШЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Преобразуем массив строк в стрим:

```
words.stream()
        .map(word -> word.split(""))
        .map(Arrays::stream)
        .distinct()
        .collect(toList());
```

FLATMAP - РЕШЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Преобразуем массив строк в стрим:

```
words.stream()
        .map(word -> word.split(""))
        .map(Arrays::stream)
        .distinct()
        .collect(toList());
```

Снова проблема: второй map вернул Stream<Stream<String>> а не Stream<String>

FLATMAP - РЕШЕНИЕ

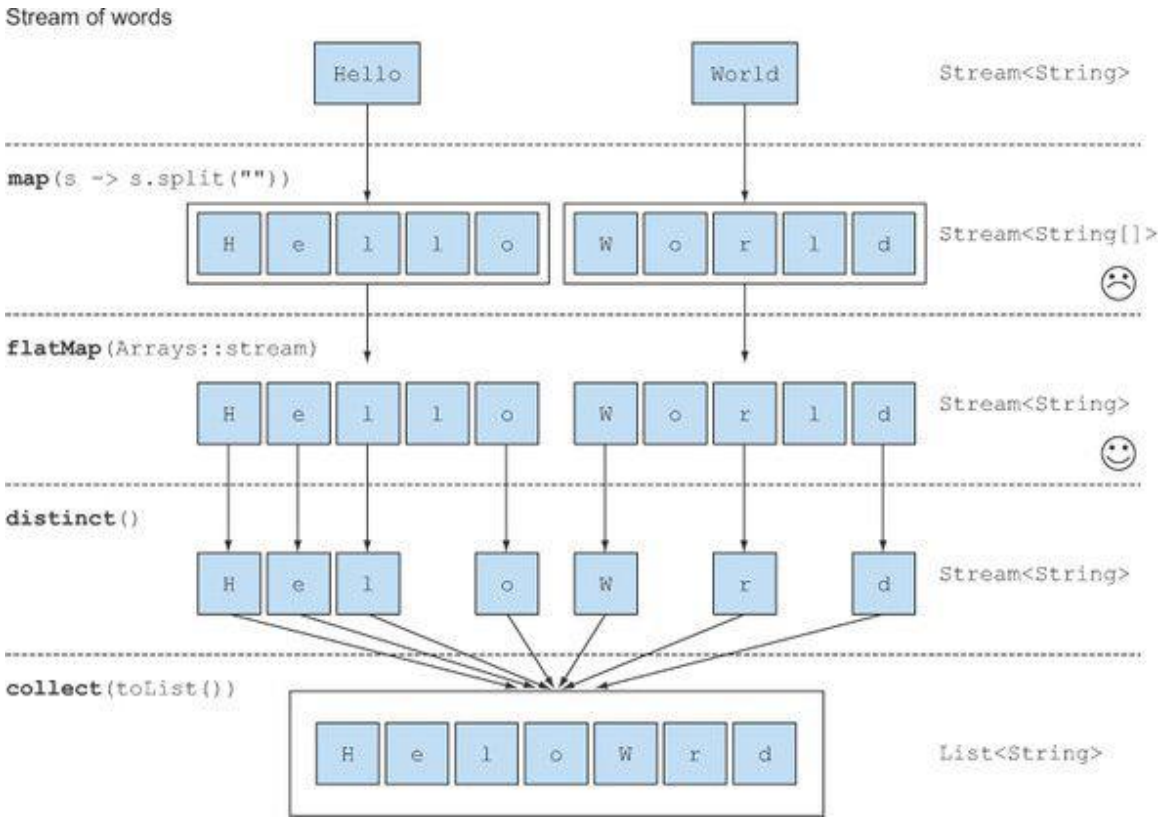
СБЕРБАНК ТЕХНОЛОГИИ

```
words.stream()
        .map(word -> word.split(""))
        .flatMap(Arrays::stream)
        .distinct()
        .collect(toList());
```

flatMap объединяет множество стримов в один

FLATMAP – ВИЗУАЛИЗАЦИЯ РЕШЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ



ОСНОВНЫЕ TERMINATE ОПЕРАЦИИ - REDUCE

СБЕРБАНК ТЕХНОЛОГИИ

Задача – просуммировать элементы коллекции

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

ОСНОВНЫЕ TERMINATE ОПЕРАЦИИ - FOREACH

СБЕРБАНК ТЕХНОЛОГИИ

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
        .filter(i -> i % 2 == 0)
        .distinct()
        .forEach(System.out::println);
```

РАСШИРЕННЫЙ REDUCTIONS С COLLECTORS

СБЕРБАНК ТЕХНОЛОГИИ

Статические методы класса java.util.stream.Collectors позволяют решать следующие терминальные задачи стримов:

- Схлопывания элементов стрима в одно значение, нахождение суммы
- Группировка элементов по произвольным критериям
- Разбиение элементов по группам

COLLECTOR: REDUCING

СБЕРБАНК ТЕХНОЛОГИИ

Можно более лаконично решить задачи нахождения максимума и минимума:

```
Optional<Dish> mostCalorieDish = menu.stream()
        .collect(maxBy(Comparator.comparingInt(Dish::getCalories)));
```

COLLECTOR: REDUCING

СБЕРБАНК ТЕХНОЛОГИИ

Можно более лаконично решить задачи нахождения максимума и минимума:

```
Optional<Dish> mostCalorieDish = menu.stream()
        .collect(maxBy(Comparator.comparingInt(Dish::getCalories)));
```

Получить статистику:

```
IntSummaryStatistics menuStatistics = menu.stream()
        .collect(summarizingInt(Dish::getCalories));

menuStatistics.getAverage();
menuStatistics.getCount();
menuStatistics.getMax();
menuStatistics.getMin();
menuStatistics.getSum();
```

COLLECTOR: ГРУППИРОВКА

СБЕРБАНК ТЕХНОЛОГИИ

Задача: вы хотите сгруппировать блюда в меню в соответствии с их типом (мясо, рыба, другое)

COLLECTOR: ГРУППИРОВКА

СБЕРБАНК ТЕХНОЛОГИИ

Задача: вы хотите сгруппировать блюда в меню в соответствии с их типом (мясо, рыба, другое)

```
Map<Dish.Type, List<Dish>> dishesByType =
        menu.stream().collect(groupingBy(Dish::getType));
```

COLLECTOR: ГРУППИРОВКА

СБЕРБАНК ТЕХНОЛОГИИ

Задача: вы хотите сгруппировать блюда в меню в соответствии с их типом (мясо, рыба, другое)

```
Map<Dish.Type, List<Dish>> dishesByType =
        menu.stream().collect(groupingBy(Dish::getType));
```

На выходе получим что-то вроде:

{FISH=[prawns, salmon], OTHER=[french fries, rice, season fruit, pizza], MEAT=[pork, beef, chicken]}

COLLECTOR: РАЗБИЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Разбиение является частным случаем группировки.

Разбивает на две коллекции одна та что соответствует предикату, другая - нет

```
Map<Boolean, List<Dish>> partitionedMenu =
        menu.stream().collect(partitioningBy(Dish::isVegetarian));
```

COLLECTOR: РАЗБИЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Разбиение является частным случаем группировки.

Разбивает на две коллекции одна та что соответствует предикату, другая - нет

```
Map<Boolean, List<Dish>> partitionedMenu =
        menu.stream().collect(partitioningBy(Dish::isVegetarian));
```

На выходе получим что-то вроде:

{false=[pork, beef, chicken, prawns, salmon], true=[french fries, rice, season fruit, pizza]}

ЧИСЛОВЫЕ СТРИМЫ

СБЕРБАНК ТЕХНОЛОГИИ

В предыдущем коде есть проблема производительности:

```
int calories = menu.stream()
        .map(Dish::getCalories)
        .reduce(0, Integer::sum);
```

ЧИСЛОВЫЕ СТРИМЫ

СБЕРБАНК ТЕХНОЛОГИИ

В предыдущем коде есть проблема производительности:

```
int calories = menu.stream()
        .map(Dish::getCalories)
        .reduce(0, Integer::sum);
```

Boxing/unboxing

ЧИСЛОВЫЕ СТРИМЫ

СБЕРБАНК ТЕХНОЛОГИИ

В предыдущем коде есть проблема производительности:

```
int calories = menu.stream()
        .map(Dish::getCalories)
        .reduce(0, Integer::sum);
```

Boxing/unboxing

Решение – numeric streams

```
int calories = menu.stream()
        .mapToInt(Dish::getCalories)
        .sum();
```

STREAM API ИТОГИ

СБЕРБАНК ТЕХНОЛОГИИ

- Stream – это последовательность элементов из некоторого источника, который поддерживает процессинговые операции
- Stream использует концепцию внутреннего итерирования
- Существует два типа операций со стримом: промежуточные и терминальные
- Промежуточные операции (filter, map, и др) возвращают Stream и могут быть объединены в pipeline, но не могут продуцировать результат
- Терминальные операции (forEach, count, collect) не возвращают Stream и могут возвращать результат
- Элементы стрима обрабатываются только по запросу
- Collectors содержит множество удобных терминальных методов

ДЕФОЛТНЫЕ МЕТОДЫ

СБЕРБАНК ТЕХНОЛОГИИ

Как решить задачу расширения интерфейсов, которые уже широко используются?

ДЕФОЛТНЫЕ МЕТОДЫ

СБЕРБАНК ТЕХНОЛОГИИ

Во-первых java 8 разрешает объявлять статические методы в интерфейсах.

Во-вторых java 8 ввела понятие default метода интерфейса.

ДЕФОЛТНЫЕ МЕТОДЫ

СБЕРБАНК ТЕХНОЛОГИИ

Вспомним пример

```
List<Apple> heavyApples1 = inventory.stream()
                .filter((Apple a) -> a.getWeight() > 150)
                .collect(toList());

List<Apple> heavyApples2 = inventory.parallelStream()
        .filter((Apple a) -> a.getWeight() > 150)
        .collect(toList());
```

ДЕФОЛТНЫЕ МЕТОДЫ

СБЕРБАНК ТЕХНОЛОГИИ

Вспомним пример

```
List<Apple> heavyApples1 = inventory.stream()
                .filter((Apple a) -> a.getWeight() > 150)
                .collect(toList());

List<Apple> heavyApples2 = inventory.parallelStream()
        .filter((Apple a) -> a.getWeight() > 150)
        .collect(toList());
```

stream() и parallelStream() были добавлены как дефолтные методы.

ОБЪЯВЛЕНИЕ

СБЕРБАНК ТЕХНОЛОГИИ

Новое ключевое слово языка default

```
public interface Collection<E> extends Iterable<E> {
            default Stream<E> stream() {
                return StreamSupport.stream(spliterator(), false);
            }
}
```

АБСТРАКТНЫЙ КЛАСС

СБЕРБАНК ТЕХНОЛОГИИ

Какая теперь разница между абстрактным классом и интерфейсом:

- Класс может наследовать только один абстрактный класс, но может реализовывать множество интерфейсов
- Абстрактный класс может хранить состояние через поля, а интерфейс - нет

НОВЫЙ ИНТЕРФЕЙС

СБЕРБАНК ТЕХНОЛОГИИ

Есть возможность создать свой интерфейс с дефолтными методами.

Зачем?

НОВЫЙ ИНТЕРФЕЙС

СБЕРБАНК ТЕХНОЛОГИИ

Есть возможность создать свой интерфейс с дефолтными методами.

Зачем?

- Опциональный метод
- Множественное наследование

ОПЦИОНАЛЬНЫЙ МЕТОД

СБЕРБАНК ТЕХНОЛОГИИ

```
public interface Iterator<T> {
    boolean hasNext();
    T next();
    default void remove() {
        throw new UnsupportedOperationException();
    }
}
```

ДЕФОЛТНЫЕ МЕТОДЫ - ИТОГИ

СБЕРБАНК ТЕХНОЛОГИИ

1. Интерфейсы в java 8 могут иметь содержимое через статические и дефолтные методы
2. Дефолтные методы позволяют разрабатывать обратно совместимые интерфейсы
3. Дефолтные методы могут использоваться в качестве опционального метода

ЛИТЕРАТУРА

СБЕРБАНК ТЕХНОЛОГИИ

www.kahkeshan.com/Source/introduceBook/d77268a4-568a-40de-9d7d-e2f9eaf56d92