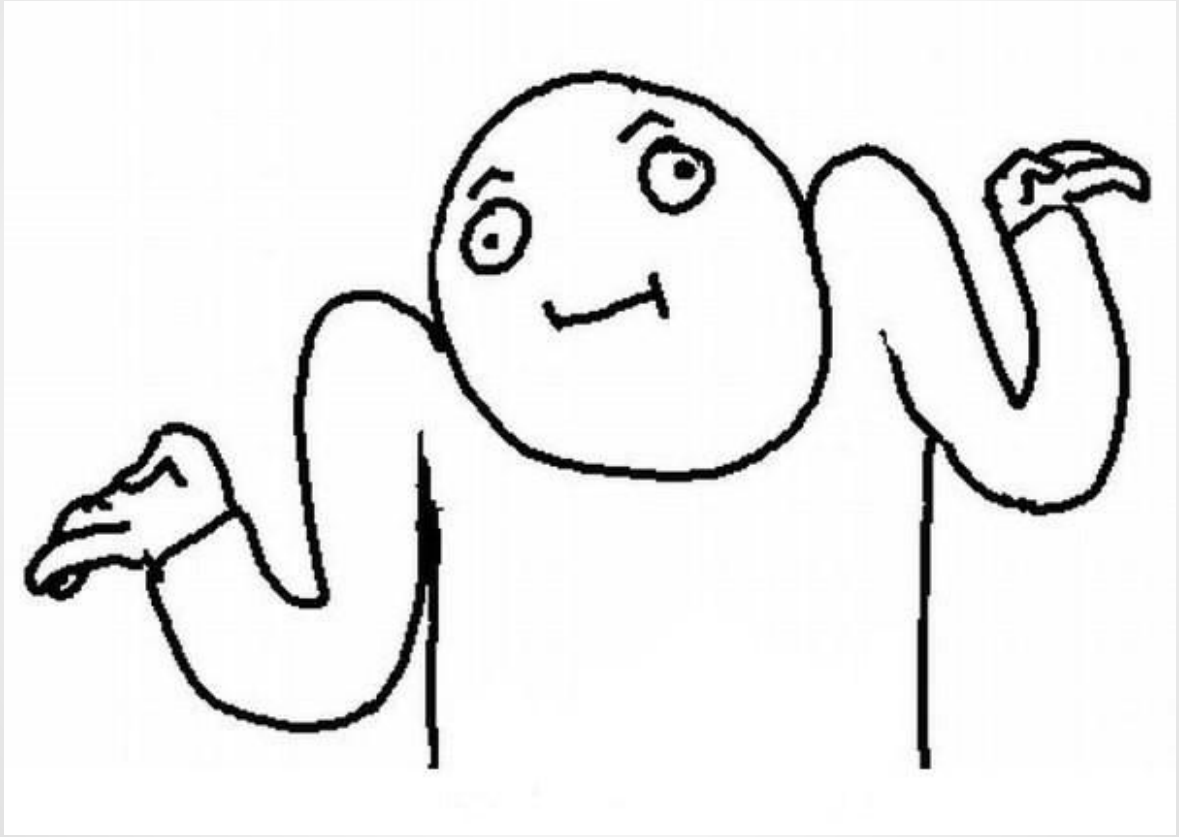
8
JAVA

Java 8 example

```
public static <T, U, A, R> Collector<T, ?, R> mapping(
             Function<? super T, ? extends U> mapper,
             Collector<? super U, A, R> downstream) {
    BiConsumer<A, ? super U> downstreamAccumulator =  downstream.accumulator();
     return new CollectorImpl<>(downstream.supplier(),
                    (r, t) -> downstreamAccumulator.accept(r, mapper.apply(t)),
                    downstream.combiner(), downstream.finisher(),
                    downstream.characteristics());
}

public static Collector<CharSequence, ?, String> joining() {
     return new CollectorImpl<CharSequence, StringBuilder, String>(
             StringBuilder::new, StringBuilder::append,
             (r1, r2) -> { r1.append(r2); return r1; },
             StringBuilder::toString, CH_NOID);
}
```

Зачем ?



Краткий список нововведений в Java 8

- Методы по умолчанию
- Лямбда выражения
- Ссылки на методы
- Новый API для потоков
- Повторяемые аннотации
- Рефлексия для параметров методов
- Поддержка Unicode 6.2.0
- Добавлен новый API для Calendar и Locale
- Новый API Date/Time;
- Новый движок JavaScript Nashorn
- Удален PermGen
- Добавлены новые классы для работы с многопоточностью
- ...

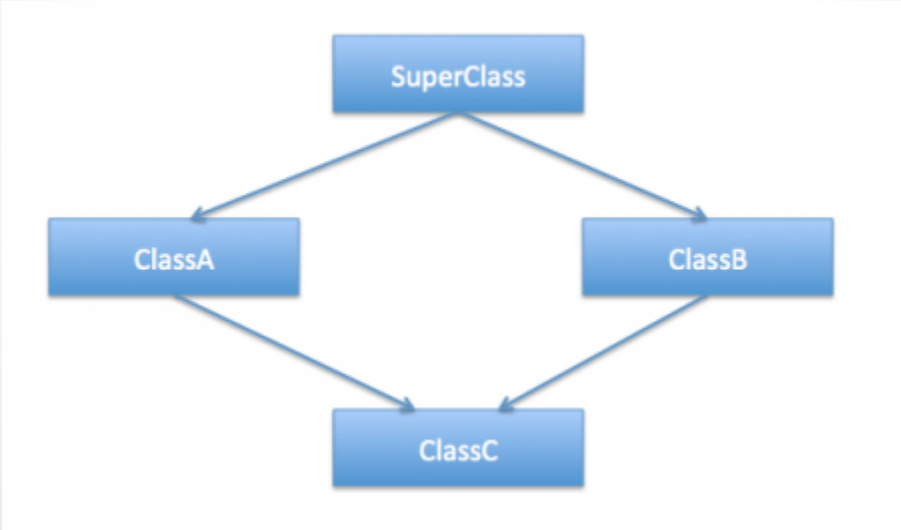
Default Methods (виртуальные методы расширения)

Виртуальные методы расширения это методы, которые можно добавить в существующие интерфейсы и предоставить реализацию по умолчанию этих методов, при этом классы-реализации не потребуют перекомпиляции и будут работать как и работали раньше.

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

Ромбовидное наследование

Ромбовидное наследование — ситуация в объектно-ориентированных языках программирования с поддержкой множественного наследования, когда два класса B и C наследуют от A, а класс D наследует от обоих классов B и C.



Наследование интерфейсов с методами по умолчанию (1)

```
public interface A {
    default void print() {
     System.out.println("interface A");
    }
}
```

```
public class Test
     implements A { }
```

```
public static void main(String[] args) {
     new Test().print();    //interface A
}
```

Наследование интерфейсов с методами по умолчанию (2)

```
public interface A {
    default void print() {
  System.out.println("interface A");
    }
}

public interface B {
    default void print() {
  System.out.println("interface B");
    }
}
```

```
public class Test implements A, B{
    @Override
    public void print() {
  System.out.println("class Test");
    }
}
```

```
public static void main(String[] args) {
    new Test().print();
}
```

??? // class Test

Наследование интерфейсов с методами по умолчанию (3)

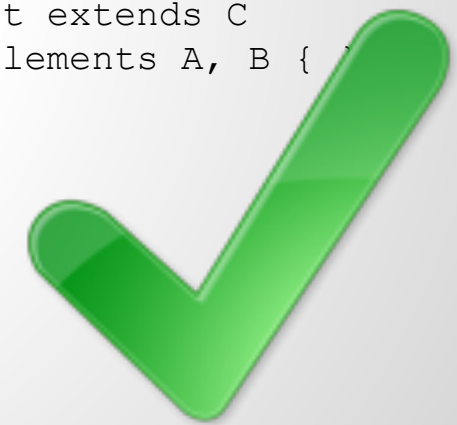
```
public interface A {
    default void print() {
  System.out.println("interface A");
    }
}

public interface B {
    default void print() {
  System.out.println("interface B");
    }
}
```

```
public class C implements A {
    @Override
    public void print() {
        System.out.println("class C");
    }
}

public class Test extends C
             implements A, B {
```

```
public static void main(String[] args) {
    new Test().print(); //class C
}
```



Наследование интерфейсов с методами по умолчанию (4)

```
public interface A {
    default void print() {
   System.out.println("interface A");
    }
}

public interface B {
    default void print() {
  System.out.println("interface B");
    }
}

public static void main(String[] args) {
    new Test().print();
}
```

```
public class C implements A { }

public class Test extends C
                  implements B { }
```

Наследование интерфейсов с методами по умолчанию (5)

```
public interface A {
    default void print() {
  System.out.println("interface A");
    }
}

public interface B extends A {
    default void print() {
  System.out.println("interface B");
    }
}


public static void main(String[] args) {
    new Test().print();    //interface B
}
```

```
public class Test implements A, B { }
```

Наследование интерфейсов с методами по умолчанию (6)

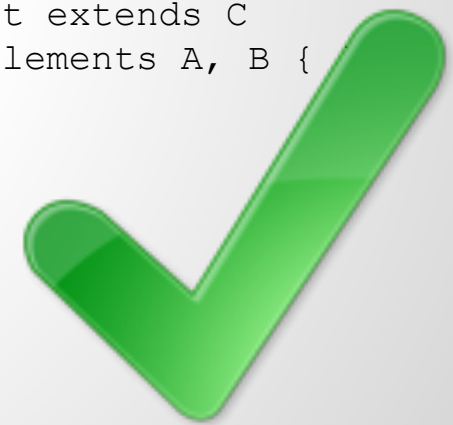
```
public interface A {
    default void print() {
  System.out.println("interface A");
    }
}

public interface B extends A {
    default void print() {
  System.out.println("interface B");
    }
}
```

```
public class C implements A {
    @Override
    public void print() {
        System.out.println("class C");
    }
}

public class Test extends C
             implements A, B {
```

```
public static void main(String[] args) {
    new Test().print();      //class C
}
```



Наследование интерфейсов с методами по умолчанию (7)

```
public interface A {
    void print();
}

public interface B {
    default void print() {
  System.out.println("interface B");
    }
}
```

```
public class Test implements A, B {
B {@Override
    public void print() {
        B.super.print();
    }
}
```

```
public static void main(String[] args) {
    new Test().print();    //interface B
}
```

Наследование интерфейсов с методами по умолчанию (итоги)

- Default методы определяются в интерфейсе и могут иметь реализацию;
- Если в супер классе существует переопределение default метода - компилятор выбирает эту реализацию;
- Если один из интерфейсов наследуется от второго интерфейса и переопределяет default метод - компилятор выбирает наиболее специфическую реализацию;
- Если реализаций default метода несколько и невозможно определить наиболее специфическую из них - необходимо явно переопределить данный метод;

lambda expressions

Пример lambda выражения

```
List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");

Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
});
```

```
Collections.sort(names, (String a, String b) -> {
    return b.compareTo(a);
});

Collections.sort(names, (String a, String b) -> b.compareTo(a));

Collections.sort(names, (a, b) -> b.compareTo(a));
```

Определение lambda expressions

- Лямбда-выражение является блоком кода с параметрами.
- Используются лямбда-выражение, когда необходимо выполнить блок кода в более поздний момент времени.
- Лямбда-выражения могут быть преобразованы в функциональные интерфейсы.
- Лямбда-выражения имеют доступ к `final` переменным из охватывающей области видимости.
- Ссылки на метод и конструктор ссылаются на методы или конструкторы без их вызова.

Функциональный интерфейс

Каждой лямбде соответствует тип, представленный интерфейсом. Так называемый *функциональный интерфейс* должен содержать **ровно один абстрактный метод**. Каждое лямбда-выражение этого типа будет сопоставлено объявленному методу.

```
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}
```

```
Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
Converter<String, Integer> c = Integer::valueOf;
Converter<byte[], String> c = String::new;
```

Является ли описанный интерфейс функциональным ?

```
public interface Function<T, R> {
    R doSomething(T t);

    default <V> Function<T, V> andThen(
        Function<? super R, ? extends V> after) {
            return (T t) -> after.doSomething(doSomething
        }
}
```

Область действия лямбда выражений

```
final int num = 1;
Converter<Integer, String> stringConverter =
     (from) -> String.valueOf(from + num);
stringConverter.convert(2);
```

```
int num = 1;
Converter<Integer, String> stringConverter =
     (from) -> String.valueOf(from + num);
stringConverter.convert(2);
```

```
int num = 1;
Converter<Integer, String> stringConverter =
      (from) -> String.valueOf(from + num);
num = 3;
```

Доступ к полям и статическим переменным (и методам)

Доступ к default методам реализуемого интерфейса

Встроенные функциональные интерфейсы

```
Predicate<String> predicate = (s) -> s.length() > 0;


Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.andThen(String::valueOf);

Supplier<Person> personSupplier = Person::new;
Person person = personSupplier.get();


Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);



Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);
```

...

java.util.Stream

stream = набор значений (НЕ множество)

stream - это НЕ структура данных

все операции - ленивые (lazy)

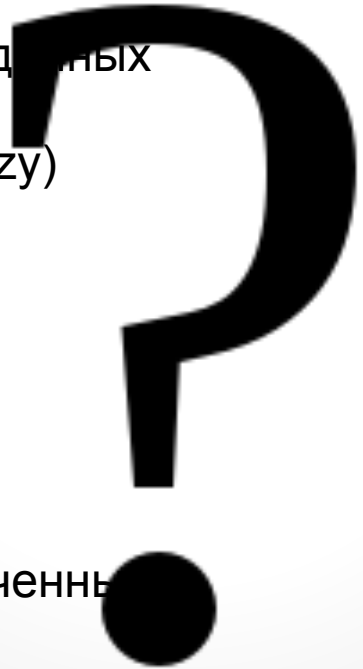
может быть бесконечным

не изменяет источник

одноразовый

упорядоченный/неупорядоченный

параллельный/последовательный



Java < 8

```
public void printGroups (List <People > people ) {
     Set <Group > groups = new HashSet < >();
     for ( People p : people ) {
          if (p. getAge () >= 65)
               groups .add (p. getGroup ());
     }
     List <Group > sorted = new ArrayList <>( groups );
     Collections . sort ( sorted , new Comparator <Group >() {
          public int compare ( Group a, Group b) {
               return Integer . compare (a. getSize (), b. getSize ())
          }
     });
     for ( Group g : sorted ) System .out . println (g. getName ());
}
```

Java 8

```
public void printGroups (List < People > people ) {
    people . stream ()
        .filter (p -> p. getAge () > 65)
        .map (p -> p. getGroup ())
        .distinct ()
        .sorted ( comparing (g -> g. getSize ()))
        .map (g -> g. getName ())
        .forEach (System.out::println);
}
```

Stream pipeline

a source: `Source` -> `Stream`

intermediate operations: `Stream` -> `Stream`

…

a terminal operation: `Stream` -> `Result`

Stream sources

- Коллекции

  ```
  Collection.stream(); Collection.parallelStream();
  ```

- Утилиты

  ```
  Arrays.stream(T[] array); Stream.of(T... values);
  IntStream.range(int startInclusive, int endExclusive);
  ```

- Генераторы:

  ```
  Stream.iterate(); Stream.generate();
  ```

- Прочее:

  ```
  bufferedReader.lines();  CharSequence.chars()
  ```

Intermediate operations

- filter - фильтрация данных
- map - преобразование с одного типа в другой
- flatMap - преобразование из одного типа в другой, при этом также возможно изменение количества элементов
- peek - произвести действие над элементами, не изменяя их
- sorted - сортировка значений
- limit - ограничение количества элементов в потоке
- skip - пропустить указанное количество элементов
- distinct - удалить повторяющиеся элементы

- sequential - сделать поток последовательным
- parallel - сделать поток параллельным
- unordered - сделать поток неупорядоченным

Terminal operation

- Терминальная операция возвращает конечный результат
- Только в момент вызова терминальной операции происходит вся обработка данных

- Группы операций:
  - итерация: `forEach, forEachOrdered`;
  - поиск: `findFirst, findAny`;
  - проверка: `allMatch, anyMatch, noneMatch`;
  - агрегаторы:
    - `collectors`
    - `reduction`

Операции с бесконечными потоками

- Некоторые операции могут “бросить” поток
- `find*, *Match, limit`

```
int v = Stream . iterate (1, i -> i +1)
    .filter ( i % 2 == 0)
    .findFirst (). get ();
```

Итерация

- ```
  IntStream . range (0, 100)
      . forEach ( System .out :: println );
  ```

- ```
  Iterator < Integer > =
       Stream . iterate (0, i -> i + 1)
        . limit (100)
        . iterator ();
  ```

Reduction

```
Stream <T> {
…
    <U> U reduce (U identity ,
    BiFunction <U,T,U> accumulator ,
    BinaryOperator <U> combiner )
...
}

Stream < Integer > s;
Integer sum = s. reduce (0, (x, y) -> x + y);
```

java.util.stream.Collectors

```
toList () => List
toSet () => Set

toCollection ( Supplier < Collection <T >>) => Collection <T>

partitioningBy ( Predicate <T >) => Map < Boolean , List <T>>
groupingBy ( Function <T,K >) => Map <K, List <T>>>

toMap ( Function <T,K>, Function <T,U >) => Map <K,U>
```

java.util.stream.Collectors

```
String [] a = new String []{ "a", "b", "c"};
```

⇓

```
"a,b,c"
```

```
Arrays
   . stream (a)
   . collect ( Collectors.joining (","));
```

Параллелизм

- Многие источники хорошо делятся на части
- Многие операции хорошо параллелизуются
- Библиотека делает всю основную работу сама
- Испозуется `ForkJoinPool`
- Нужно явно просить библиотеку

```
int v = list .parallelStream ()
    . reduce ( Math :: max)
    . get ();
```

Почему нельзя всегда использовать параллелизм ?

Выигрыш очень сильно зависит от:

- *N* - количество элементов в источнике
- *Q* - стоимость операции над элементом
- *P* - доступного параллелизма на машине
- *C* - количество конкурентных клиентов

- точно знаем *N*
- хорошо представляем *P*
- примерно знаем *C*
- *Q* - оценить практически невозможно

Stream perfomance

```
List<Integer> list = IntStream.range(0, COUNT).
                        boxed().collect(Collectors.toList());
```

Old style

```
List<Integer> tmp = new ArrayList<>();
for (Integer integer : list) {
   if ((integer % 255) == 0) {
        tmp.add(integer);
   }
}
```

Stream perfomance

```
List<Integer> list = IntStream.range(0, COUNT).
                        boxed().collect(Collectors.toList());
```

Ordered stream

```
list.stream()
        .filter(x -> (x % 255) == 0)
        .collect(Collectors.toList());
```

Stream perfomance

```
List<Integer> list = IntStream.range(0, COUNT).
                        boxed().collect(Collectors.toList());
```

Unordered stream

```
list.stream()
        .unordered()
        .filter(x -> (x % 255) == 0)
        .collect(Collectors.toList());
```

Stream perfomance

```
List<Integer> list = IntStream.range(0, COUNT).
                        boxed().collect(Collectors.toList());
```

Parallel ordered stream

```
list.stream()
          .parallel()
          .filter(x -> (x % 255) == 0)
          .collect(Collectors.toList());
```

Stream perfomance

```
List<Integer> list = IntStream.range(0, COUNT).
                        boxed().collect(Collectors.toList());
```

Parallel unordered stream

```
list.stream()
          .parallel()
         .unordered()
          .filter(x -> (x % 255) == 0)
          .collect(Collectors.toList());
```

Stream perfomance (result table)

Тип обработки	10 миллионов значений	100 значений
Old style	18	33514
stream/ordered	17	23292
stream/unordered	16	33387
parallel/ordered	34	5412
parallel/unordered	36	8429

throughput/sec

Спасибо за внимание