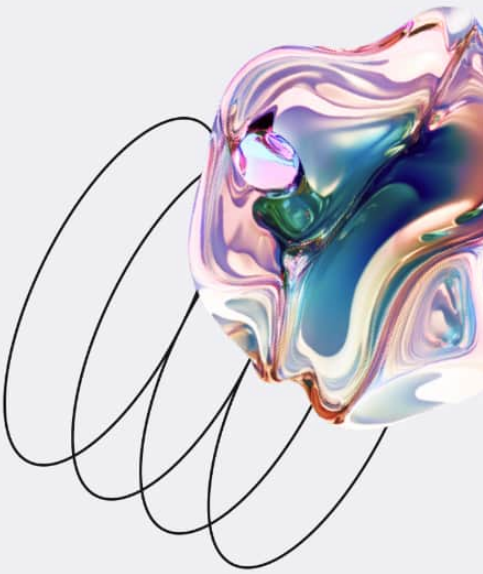
GeekBrains



Лямбды и Stream API

Java Junior

Оглавление

Введение

Всем привет! Добро пожаловать на курс Java Junior. На этом курсе мы расширим ваши знания до уровня Junior!

Чем мы будем заниматься и что изучать?

В процессе обучения вы научитесь понимать и применять на практике лямбды и Stream API, а также Reflection API для получения информации о классах и методах во время выполнения программы. Вы сможете сериализовывать и десериализовывать объекты с использованием различных форматов, таких как JSON и XML. Кроме того, вы научитесь работать с базами данных и использовать различные инструменты для взаимодействия с ними, например, JDBC. И, наконец, вы сможете разрабатывать клиент-серверные приложения, понимая принципы работы протоколов и взаимодействия между клиентом и сервером.

Термины, используемые в лекции

Лямбда — это функция, которая принимает один аргумент и возвращает значение. Она используется для создания анонимных функций в языке программирования Java. Лямбда выражения позволяют создавать функции без создания отдельного класса или метода. Они используются для обработки коллекций данных, отправки событий в обработчики и других задач.

Stream API - это библиотека для работы с потоками данных в Java. Она предоставляет множество методов для обработки коллекций, таких как фильтрация, сортировка, преобразование и т.д. Stream API позволяет писать более компактный и эффективный код, а также улучшает его читаемость.

Лямбда выражения

Начнём мы сегодня с лямбды! Появилась она в восьмой версии Java и немного разделила программистов на тех, кто разобрался и использует все ее возможности и тех, кто или взял примитивное определение необходимое для работы или не использует вообще! Итак что такое лямбда, и где ей можно воспользоваться?

Лямбда - это функция, описывающая обработку данных, и оно очень похоже на метод. Однако его можно передавать как аргумент. Существует множество классов, методы которых ожидают именно лямбда-выражения в качестве аргументов. Наиболее распространенные из них реализованы в интерфейсе Stream API. Работа этого интерфейса заключается в подборе функций для обработки имеющихся данных, поэтому мы начнем наше обучение с рассмотрения лямбда-выражений и только потом перейдем к изучению Stream API.

Давайте создадим простой класс. В качестве точки входа используем метод main, реализуем интерфейс PlainInterface и создадим метод action непосредственно в классе. В Java такое решение используется нечасто, обычно вместо этого мы добавляем статичный метод.

```
public class PlainClass {

  public static void main(String[] args) {
    PlainInterface anInterface = new PlainInterface() {
      @Override
      public String action(int x, int y) {
        return String.valueOf(x+y);
      }
    };
    System.out.println(anInterface.action(5, 5));
  }
}
interface PlainInterface{String action(int x, int y);}
```

Но давайте сначала перепишем реализацию с использованием лямбда выражения.

```
public class PlainClass {

   public static void main(String[] args) {
      PlainInterface anInterface = (x, y) -> String.valueOf(x+y);
      System.out.println(anInterface.action(5, 5));
   }
}

@FunctionalInterface
interface PlainInterface{String action(int x, int y);}
```

Изменился синтаксис, и код стал проще. Не нужно создавать экземпляр класса, не нужна аннотация оверрайд и, если посмотреть внимательней, мы не передаем типы параметров и не указываем ключевое слово return. **Всё это делает компилятор!** Ну и если говорить о мелочах, не нужно оборачивать реализацию в фигурные скобки и завершать её точкой с запятой. Однако фигурные скобки, ограничивающие блок кода, не нужны только если реализация умещается в одну строку. Если же вы захотите многострочную реализацию, то и фигурные скобки и return вернутся на свои места. Это чуть больше похоже на классическую реализацию, однако всё же короче и лаконичней! И ещё обращу ваше внимание на новую аннотацию @FunctionalInterface. Она проверяет, является ли интерфейс функциональным, то есть описывает всего один метод. Это обязательно для лямбда выражений и если это не так, аннотация просто не даст коду собраться.

```
public class PlainClass {

   public static void main(String[] args) {
      PlainInterface anInterface = (x, y) -> {
         String str = String.valueOf(x+y);
         return str+"!";
      };
```

```
        System.out.println(anInterface.action(5, 5));
    }
}
@FunctionalInterface
interface PlainInterface{String action(int x, int y);}
```

А давайте попробуем создать несколько реализаций!

```
public class PlainClass {
    static PlainInterface anInterface;

    public static void main(String[] args) {
        anInterface = (x, y) -> String.valueOf(x+y);
        print(5, 5);
        anInterface = (x, y) -> String.valueOf(x-y);
        print(5, 5);
        anInterface = (x, y) -> String.valueOf(x*y);
        print(5, 5);
    }
    private static void print(int x, int y){
        System.out.println(anInterface.action(x, y));
    }
}
@FunctionalInterface
interface PlainInterface{String action(int x, int y);}
```

Для демонстрации я создал метод print, который выводит в консоль результаты работы нашего лямбда-выражения. В методе main я трижды переопределил поведение лямбда-выражения и трижды вызвал метод print с одинаковыми аргументами. Результаты соответствуют ожидаемому поведению: сумма, разность или произведение. Это простой и наглядный пример переопределения метода anInterface непосредственно в коде! Давайте теперь изменим код еще немного.

```
public class PlainClass {
   static PlainInterface anInterface;
   static PlainInterface2 anInterface2;

   public static void main(String[] args) {
      anInterface = (x, y) -> String.valueOf(x+y);
      System.out.println(anInterface.action(5,5));
      anInterface2 = Integer::compare;
      System.out.println(anInterface2.action(5,15));
   }
}
@FunctionalInterface
interface PlainInterface{ String action(int x, int y);}
@FunctionalInterface
interface PlainInterface2{ int action(int x, int y);}
```

Я добавил еще один интерфейс с другой сигнатурой метода и реализовал его по-другому. В этой реализации нет переменных, только имя класса и имя метода, разделенные двумя двоеточиями. Как это работает? Метод compare класса Integer принимает два int параметра и возвращает int значение. Если бы мы описали это с помощью обычного лямбда-выражения, то получили бы следующий код.

```
PlainInterface2 interface2 = (x, y) -> Integer.compare(x, y);
```

Мы создали своего рода оболочку для метода compare, и лямбда снова предлагает нам лаконичное решение. Всё, что нам нужно сделать в реализации — это указать имя класса, а через двоеточие — имя метода. Компилятор сам поймёт, что методу

нужны два параметра, а мы передаем ему именно два параметра нужных типов, и метод возвращает то, что требует наш функциональный интерфейс. Вот откуда у нас такой простой синтаксис. У лямбда-выражений много других возможностей, но я бы хотел завершить их обсуждение здесь. Конечно, более глубокие знания не помешают, но очень важно понять всё то, что мы уже обсудили. Я бы еще раз определил лямбда-выражения с учетом приведенных нами примеров. Лямбда создаются на основе функциональных интерфейсов и позволяют переопределить поведение непосредственно в коде, упрощая синтаксис. Они значительно облегчают написание кода, делают его короче и понятнее, напоминая нам о синтаксическом сахаре. Лямбда позволяют использовать преимущества функционального программирования на полностью объектно-ориентированном языке Java.

- Лямбды формируются на основе функционального интерфейса и могут переопределять поведение прямо в коде с упрощенным синтаксисом.
- Лямбды сильно упрощают программирование, делают код проще и короче! И этим напоминают синтаксический сахар.
- Лямбды, являясь функциями, позволяют использовать в полностью объектно-ориентированном языке программирования Java плюсы из функционального программирования. В частности передавать и хранить сами реализации методов!

У лямбда-выражений масса преимуществ, но некоторые из них не так важны для прикладного программиста, а другие гораздо проще изучить на практических примерах. На семинарах у вас будут практические задания по функциональным интерфейсам и их реализации (что и есть лямбда-выражения!), и вы сможете самостоятельно пощупать и опробовать их в деле. А сейчас я бы хотел перейти к следующей теме — Stream API.

Stream API

Название интерфейса намекает на потоки ввода-вывода или на многопоточность, но в Stream API «поток» — это просто набор данных, а интерфейс позволяет нам обрабатывать эти данные с помощью нужных нам функций. Функции передаются в виде лямбда-выражений в качестве параметров, а обработка выполняется последовательно. В Stream API такая последовательность называется «конвейером». Подбирая нужные функции для обработки данных, мы строим конвейер — гибкий, быстрый и легко расширяемый. Конвейеры состоят из методов интерфейса, а сами методы делятся на три типа: генераторы, фильтры и коллекционеры. Но давайте взглянем на простой пример.

```
List<String> myList = Arrays.asList("Привет","мир","!","Я","родился","!");
myList.stream().filter(s -> s.length()>4).forEach(System.out::println);
```

Первая строка довольно проста и просто создает список строк myList с использованием метода asList класса-помощника Arrays. Но вторая строка интереснее! В Java 8 списки и множества получили новый метод stream(). Его вызов создает обычный поток данных. Можно назвать этот метод “генератором потока”. Теперь у нас есть поток, и мы можем преобразовать его в конвейер, добавив фильтр! Метод filter() является фильтром! Точнее, их правильнее называть промежуточными методами или методами конвейера, поскольку не все из них имеют слово “фильтр” в названии! Как я говорил ранее, в качестве параметра конвейерному методу я передал лямбда-выражение. Мне не нужно объявлять функциональный интерфейс или выполнять еще какие-либо действия, сам метод ожидает лямбда-выражение в качестве параметра! У него уже есть данные, остается описать, что с ними делать. Однако для лямбда-выражений существуют два ограничения. Во-первых, они должны быть невмешивающимися (non-interfering), т. е. не менять исходных данных. Во-вторых, они не должны запоминать состояние (stateless), т.е. не зависеть от порядка выполнения, от внешних переменных и от всего внешнего пространства в целом! В этом примере я создал поток из списка, но можно создавать потоки и другими методами.

Пример:

```
Arrays.asList(1, 2, 3).stream();

Метод asList теперь сам умеет создавать потоки данных!

Stream.of(3, 2, 1);
```

Вы можете создавать потоки с помощью Stream.of, что иногда может быть более удобным. Однако это еще не все, поскольку Stream API также позволяет создавать специализированные потоки для работы с примитивными типами: IntStream, LongStream и DoubleStream. Поток IntStream можно использовать подобно обычному циклу for(;;), используя метод range.

```
IntStream.range(1, 4);
```

У таких, вспомогательных, потоков есть пару дополнительных методов таких как sum() и average().

Пример:

```
IntStream.range(1, 4).average().ifPresent(System.out::println);
```

Выводит в консоли среднее значение чисел от 1 до 3. Разобравшись с источниками, переходим к следующему этапу. Важно не только создать поток, но и уметь с ним работать. Здесь мы рассмотрим конвейерные методы, созданные специально для обработки данных в потоке. И здесь есть одна особенность. Они работают по принципу “ленивой” обработки данных! Я буду обращать ваше внимание на эту особенность по мере чтения лекции, чтобы вы могли оценить все преимущества такого подхода. Один Stream может иметь сколько угодно конвейерных операций, и только одну терминальную. Терминальными я буду называть методы-коллекторы по той же причине, по которой конвейерные методы лучше называть промежуточными. Не все коллекторы содержат в своем названии слово “collect”!). Третью часть лекции мы посвятим терминальным методам, но если Stream с конвейерными операциями не завершить терминальной операцией, никаких действий выполнено не будет. Это связано с тем, что я как программист не просил программу сохранять результаты ее работы. Зачем выполнять работу, если результат не будет использован? Это одна из ключевых особенностей “ленивого”

подхода. До тех пор пока мы не перейдем к рассмотрению терминальных методов, будем использовать forEach() в качестве примера.

Пример:

```
List<String> list = Arrays.asList("Не", "заменят", "край", "родимый",
      "Никакие", "чудеса!", "Только", "здесь",  "всё", "так", "любимо",
      "Реки", "горы", "и", "леса.");
list.stream().filter(n->n.length()>4).forEach(n->System.out.print(n+" "));
```

Мы подготовили список данных в первой строке и создали поток со встроенным фильтром во второй строке. Фильтр проверяет длину строки, которая должна быть не менее четырех букв. В терминальном методе мы просто выводим результат в консоль. Результат верен только для слов длиной больше четырех символов. Однако мы написали исходный код не очень аккуратно. Обычно при использовании Stream API вызовы методов размещаются на отдельных строках.

Пример:

```
 list.stream()
      .filter(n -> n.length()>4)
      .forEach(n -> System.out.print(n+" "));
```

Вот так код выглядит лаконичней! Усложню выборку ещё одним фильтром.

```
list.stream()
      .filter(n -> n.length()>4)
      .filter(c -> c.toLowerCase().contains("a"))
      .forEach(n -> System.out.print(n+" "));
```

Мы добавили одну строку, и все слова, которые не содержат букву “a”, исчезли. Здесь есть одна особенность: в лямбде мы использовали метод toLowerCase(), который возвращает строку в нижнем регистре, но результат остался в исходном регистре. Дело в том, что filter, как и следует из его названия, не изменяет исходные

данные, а только выбирает необходимые, удовлетворяющие условию, описанному в лямбде. Это очень удобно, в частности, тем, что при написании условий можно не беспокоиться о целостности исходных данных. Следующим идет конвейерный метод skip, который позволяет пропустить заданное количество первых элементов потока

Пример:

```
list.stream()
      .skip(list.size()/2)
      .forEach(n -> System.out.print(n+" "));
```

Мы пропустили первую половину данных потока и работать начали только со второй! Давайте объединим filter и skip

Пример:

```
list.stream()
      .skip(list.size()/2)
      .filter(n -> n.length() >4)
      .filter(n -> n.toLowerCase().contains("a"))
      .forEach(n -> System.out.print(n+" "));
```

В данном случае в результате будет только слово "леса"! Ибо во второй половине текста только оно выполняет поставленные условия. Следующим конвейерным методом будет limit. Это skip наоборот! Ограничивает обработку указанным количеством первых элементов.

Пример:

```
list.stream()
      .limit(list.size()/2)
      .filter(n -> n.length() >4)
      .filter(n -> n.toLowerCase().contains("a"))
      .forEach(n -> System.out.print(n+" "));
```

Тот же код, но skip поменяем на limit. Ну и результат совсем другой! Теперь выборка из первой половины. Метод простой поэтому переходим к следующему. Distinct, тут тоже всё просто, он пропускает поток без повторов.

Пример:

```
List<String> list = Arrays.asList("а", "б", "а", "в", "а", "г", "а", "д");

list.stream()
      .distinct()
      .forEach(n -> System.out.print(n));
```

Тут всё просто - никаких параметров, и в результате мы получаем "абвгд", без повторных вхождений! Еще один конвейерный метод называется sorted и возвращает поток, отсортированный по возрастанию.

Пример:

```
myList.stream().sorted().forEach(System.out::println);
```

Это простейший случай вызова без параметров. Есть более сложный вариант с параметром! Крутость его в том, что этот параметр - компаратор, и мы сами можем управлять порядком сортировки. Также мы можем сами создать компаратор!

Пример:

```
myList.stream()
      .sorted((s, t1) -> t1.length() - s.length())
      .forEach(System.out::println);
```

Здесь я хочу обратить ваше внимание на компаратор. Его поведение определяется лямбда-выражением. В данном примере это простейшая однострочная функция, но вы не ограничены такой реализацией!

Пример:

```
myList.stream()
      .sorted((s, t1) -> {
         int tmp = t1.length() - s.length();
         if (tmp<0) return 1;
         else if (tmp>0) return -1;
         return 0;
      })
      .forEach(System.out::println);
```

Я немного усложнил компаратор! Сделал его многострочным и появились фигурные скобки и появилась необходимость писать return своими руками! Но синтаксис, даже в таком не сложном примере всё же проще и лаконичнее чем в реализации без стримов! Ещё один конвейерный метод и называется он map! Этот метод проходит по всем элементам и возвращает в поток данные изменённые по логике его лямбда выражения. Он, в соответствии концепции non-interfering не изменяет исходные данные, только данные потока.

Пример:

```
List<String> list = Arrays.asList("Привет", "Как дела?", "Пропеллер!", "никель");
list.stream().map(n -> n.length()).forEach(System.out::println);
```

Я немного усложнил компаратор, сделав его многострочным. Теперь появились фигурные скобки, и вам нужно самостоятельно написать return. Однако, даже в этом несложном примере, синтаксис остается более простым и лаконичным, чем в варианте без стримов. Следующий метод конвейера называется map. Этот метод проходит через все элементы и возвращает в поток измененные данные в соответствии с логикой его лямбда-выражения. Он не меняет исходные данные, изменяя только данные потока, в соответствии с концепцией non-interefering.

Пример:

```
List<String> myList = Arrays.asList("Привет","мир","!","Я","родился","!");
System.out.println(myList.stream().filter(s -> s.length()>4).findFirst());
```

И результатом будет строка в консоли:

Optional[Привет]

Метод вывел слово "Привет", так как это первое попавшееся в потоке слово удовлетворяющее условию. В данном случае метод findFirst() можно было бы заменить конструкцией.

Пример:

```
myList.stream()
        .filter(s -> s.length()>4)
        .limit(1)
        .forEach(n -> System.out.print(n));
```

Результат будет тот же. Но конвейер длиннее и соответственно медленнее. Есть ещё один похожий метод. findAny(). Он, как и предыдущий, возвращает один результат из потока но не обязательно первый в списке а первый обработанный. Это ещё одна отсылка к так называемой ленивой обработке.

Пример:

```
System.out.println(myList.stream().filter(s -> s.length()>4).findAny());
```

Терминальные методы findAny и findFirst возвращают результатом экземпляр класса Optional, поэтому и в консоли мы видим "Optional[Привет]". Это не всегда удобно. В основном, мы хотим увидеть массивы или списки. Для этого есть другой метод, и называется он collect! С его помощью мы можем представить данные в виде нужных нам структур данных.

Пример:

```
List<String> myList = Arrays.asList("Привет", "мир", "!", "Я", "родился", "!", "Море", "Поле");

    List<String> tmpList = myList.stream()
        .sorted((s, t1) -> {
```

```
            int tmp = t1.length() - s.length();
            if (tmp<0) return 1;
            else if (tmp>0) return -1;
            return 0;
        })
        .collect(Collectors.toList());
```

В методе collect я использую параметр toList, который является частью вспомогательного класса Collectors. Когда данные проходят через конвейер, они собираются в список для дальнейшей работы с ними.

Я рассказал вам о некоторых возможностях стримов и показал на практике, как они работают. В более сложном примере я описал каждый шаг подробно. Monthes — это просто перечисление месяцев для удобного вывода. PersonID — это простой класс, описывающий сотрудника с закрытыми полями: id, ФИО, датой рождения, зарплатой и геттерами. В методе main инициализируется список personIDS и генерируется случайная зарплата. Затем создается поток данных из списка personIDS и преобразуется в поток дат рождения с использованием лямбда-метода. Метод collect с параметром Collectors.toList собирает все полученные данные в список, который сохраняется в result.

Пример:

```
List<String> tmpList = personIDS.stream()
  .filter(n -> n.getDATE().compareTo(new Date(1995, Calendar.JANUARY, 1))>0)
  .map(PersonID::getDOB).collect(Collectors.toList());
System.out.println(tmpList);
```

В этом примере я добавил только фильтр. Я выбираю из потока только тех, кто родился после 1 января 1995 года. Из исходного списка в 12 человек были выбраны только 6. Можно усложнить задачу, добавив еще один фильтр.

Пример:

```
List<String> tmpList = personIDS.stream()
  .filter(n -> n.getDATE().compareTo(new Date(1995, Calendar.JANUARY, 1))>0)
  .sorted((a,b)-> (int) (a.getSalary()-b.getSalary()))
  .map(PersonID::getFIO).collect(Collectors.toList());
```

К фильтру и сортировке добавился еще один метод - map. Теперь мы собираем имена сотрудников и сортируем их по уровню зарплаты. Результат - список сотрудников с именами, родившимися после 1995 года, отсортированными по зарплате.

Пример:

```
List<String> tmpList = personIDS.stream()
  .filter(n -> n.getDATE().compareTo(new Date(1995, Calendar.JANUARY, 1))>0)
  .sorted((a,b)-> (int) (a.getSalary()-b.getSalary()))
  .map(n -> n.getFIO() + " (" + n.getDOB() + ") " + n.getSalary())
  .limit(5)
  .collect(Collectors.toList());
tmpList.forEach(n-> System.out.println(n));
```

На этот раз я внес небольшие изменения. В методе map я добавил метод limit(5), который ограничивает список первыми пятью элементами. Мы берем список объектов, отбираем только те, которые соответствуют определенному условию, затем сортируем полученный список по одному из полей объекта и, наконец, создаем новый список с описаниями объектов и также ограничиваем его первыми пятью элементами. Весь этот процесс можно выполнить без стримов, но Stream API и лямбда-выражения упрощают эту задачу. Лямбда-выражения позволяют по-новому подойти к написанию кода. Имея поток данных, мы можем прогнать его через различные функции, изменять его и все это в рамках компактного и понятного синтаксиса. Хотя стримы и лямбды имеют гораздо больше возможностей, но для начала работы этого будет достаточно.

Пример

```
enum Monthes{Января, Февраля, Марта, Апреля, Мая, Июня, Июля, Августа,
Сентября, Октября, Ноября, Декабря}
class PersonID {
  private final int ID;
  private final String FIO;
  private Date DOB;
  private float salary;

  PersonID(String fio, Date dob, float salary) {
    FIO = fio; DOB = dob; ID = new Random().nextInt();
    this.salary = salary;
  }
  public int getID() {return ID;}
  public String getFIO() {return FIO;}
            public String getDOB() {return ""+DOB.getDate() + " " +
Monthes.values()[DOB.getMonth()] + " " + DOB.getYear();}
  public float getSalary() {return salary;}
  public void setSalary(float salary) {this.salary = salary;}
}

public class Str {
  public static void main(String[] args) {

    List<PersonID> personIDS = Arrays.asList(
              new PersonID("Иванов И.И.", new Date(1992, Calendar.FEBRUARY, 7),
genSalary()),
                new PersonID("Петров П.В.", new Date(1987, Calendar.APRIL , 27),
genSalary()),
            new PersonID("Селиванов В.А.", new Date(1995, Calendar.AUGUST, 15),
genSalary()),
             new PersonID("Кладовцева Я.И.", new Date(1996, Calendar.JUNE, 28),
genSalary()),
          new PersonID("Стильнов В.М.", new Date(1981, Calendar.SEPTEMBER, 18),
genSalary()),
            new PersonID("Иванова С.В.", new Date(1991, Calendar.FEBRUARY, 17),
genSalary()),
            new PersonID("Одоевцева М.В.", new Date(2001, Calendar.JANUARY, 6),
genSalary()),
```

```
                new PersonID("Кузеванов А.И.", new Date(2003, Calendar.JUNE, 14), genSalary()),
                    new PersonID("Донцев Ю.Ф.", new Date(1991, Calendar.MAY, 22), genSalary()),
                new PersonID("Кривцова А.И.", new Date(1976, Calendar.DECEMBER, 4), genSalary()),
              new PersonID("Бронникова И.И.", new Date(1999, Calendar.OCTOBER, 19), genSalary()),
                new PersonID("Остафьев И.А.", new Date(1995, Calendar.FEBRUARY, 24), genSalary())
        );

                            List<String> tmpList = personIDS.stream().map(n -> n.getDOB()).collect(Collectors.toList());

      System.out.println(tmpList);

    }

    private static int genSalary(){return new Random().nextInt(63758)+16242;}
}
```

Заключение

На первом уроке нашего курса, Лямбды и Stream API, мы рассмотрели основные концепции и преимущества использования этих инструментов в разработке на Java. Лямбда-выражения упрощают написание и использование анонимных классов, а Stream API предоставляет мощный и эффективный способ обработки коллекций и работы с потоками данных.

Мы изучили различные методы работы со Stream API, такие как map, filter, reduce, forEach и другие, и научились применять их для выполнения различных операций над коллекциями и потоками. Это позволило нам значительно сократить количество кода и улучшить читаемость наших программ.

Использование лямбд и Stream API позволяет нам писать более компактный, эффективный и читаемый код, что является ключевым аспектом для достижения качества и надежности наших приложений. Это делает их незаменимыми инструментами для Java-разработчика любого уровня, включая Junior.

Также мы с вами изучили основы лямбд, Stream API и их применение в реальных проектах, что поможет нам в дальнейшем развитии наших навыков в области Java-разработки.