Освен данни повечето програмни езици позволяват като параметър на функция да се подаде и друга функция. Това е полезно в голям брой ситуации. Например при сортиране на колекция от данни може динамично да се променя критерият, по който те се сортират, без да се модифицира алгоритъмът за сортиране.

В Java тази функционалност се имплементира посредством т.нар. функционални интерфейси. Функционален интерфейс е интерфейс, който съдържа точно един абстрактен метод. По конвенция е прието те да бъдат маркирани със съответна анотация (макар че тя не е задължителна):

```
@FunctionalInterface
  interface Printable
  {
    String print(String str);
  }
```

Както вече е известно, не може да се създаде директно инстанция на този интерфейс, тъй като все още няма имплементация за метода print. За да се използва Printable, е необходимо да се създаде конкретен клас, който го имплементира:

```
class MyPrintable implements Printable
    {
      @Override
      public String print(String str)
      {
        System.out.println(str);
        return str;
      }
    }
Printable myPrintable = new MyPrintable();
```

След това инстанцията на MyPrintable може да бъде подадена на метод, който приема като параметър Printable:

```
void printHello(Printable printable)
  {
    printable.print("Hello");
  }
printHello(myPrintable);
```

Тъй като Printable съдържа единствено метода print, на практика при подаване на Printable обект като параметър се подава конкретна имплементация на функция.

Ако такава функция ще бъде използвана само на едно място в кода, е неудобно всеки път да се декларира за нея отделен клас. Java позволява такива класове да бъдат декларирани директно на мястото, където ще се използват:

```
Printable printable = new Printable()
    {
       @Override
       public String print(String str)
       {
          System.out.println(str);
          return str;
       }
    };
```

Важно е да се отбележи, че при този запис не се създава инстанция на интерфейс. Вместо това се декларира клас, който имплементира Printable и всичките му абстрактни методи. Такъв клас се нарича анонимен, тъй като не му е зададено конкретно име (каквото е MyPrintable).

От Java 8 насам съществува и по-удобен синтаксис за създаване на анонимни класове – т.нар. lambda израз:

```
Printable printable = (s) ->
    {
       System.out.println(s);
       return s;
    };
```

Особености при работа с lambda са:

- могат да се използват единствено с функционални интерфейси;
- в списъка с параметри се пропуска типа (той се подразбира от декларацията на интерфейса);
- параметрите могат да се преименуват (в случая str е прекръстен на s)
- пропуска се връщаният тип на функцията (отново той се подразбира)

Ако функцията може да бъде декларирана на един ред, могат да се пропуснат фигурните скоби и ключовата дума return. Например:

```
@FunctionalInterface
   interface Summable
   {
      int sum(int a, int b);
   }
Summable s = (a, b) -> a + b;
```

Както бе отбелязано, lambda изразите често се използват за работа с колекции. В Java за обработки на колекции от данни се използва приложно-програмният интерфейс Stream API.

Важно: Stream API да не се бърка с потоците InputStream и OutputStream, които също се наричат streams.

Първо е необходимо да се подаде колекцията на Stream чрез метода stream(). След това могат да бъдат извършвани обработки като сортиране и филтрация. Накрая, за да се възстанови колекцията, се извиква метода collect().

Следва пример със сортиране на колекция от Person по имената им и премахване на имената, които започват с „А“:

```
class Person
    {
       private String name;
       …
    }
List<Person> people = new ArrayList<>();
people.add(new Person("Ivan"));
people.add(new Person("Ana"));
people.add(new Person("Georgi"));

people = people.stream()
          .filter(p -> !p.getName().startsWith("A"))
          .sorted((p1, p2) -> p1.getName().compareTo(p2.getName()))
          .collect(Collectors.toList());
System.out.println(people); // [Georgi, Ivan]
```

Методите filter() и sorted() приемат като параметър функционалните интерфейси Predicate и Comparator. Техни анонимни инстанции се подават чрез lambda изразите.

Ако в Comparator-а ще се сравняват просто полетата на класа, може допълнително да се улесни записа, като се подаде просто референция към съответния get метод. Синтаксисът за това е:

ИмеНаКлас::имеНаМетод

Или в примера:

```
.sorted(Comparator.comparing(Person::getName))
```

Пълен списък с методите на Stream:

https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Задача:

Да се реализира софтуер за разпределяне на избиратели по изборни секции. Създайте клас Voter с полета name, city, street и address. Създайте списък от Voter. Премахнете от списъка всички избиратели, които не са от София с filter(). След това сортирайте по улица и номер на адрес. За последователно сортиране по няколко полета използвайте thenComparing():

… sorted(Comparator.comparing(MyClass::getField1).thenComparing(MyClass::getField2)) …