دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

دانشکده ریاضی و علوم کامپیوتر

برنامه‌سازی پیشرفته و کارگاه

lambda expression

استاد درس

دکتر مهدی قطعی

استاد دوم

بهنام یوسفی مهر

نگارش

کیانا رضوی و سیدآرمان حسینی

بهار ۱۴۰۳

فهرست

مقدمه 3

فیلتر کردن داده 4

راه حل اول: برای هر کدوم یه متد بنویسیم 4

راه حل دوم: اینترفیس CheckPerson 5

راه حل سوم: Lambda Expressionها 6

راه حل چهارم: اینترفیس‌های استاندارد جاوا 7

اینترفیس Consumer 8

اینترفیس Function 8

چیزی که یاد گرفتیم 10

منابع بیشتر 10

مقدمه

خیلی وقت‌ها، ما نیاز داریم با توابع مثل یک object برخورد کنیم. مثلا، شاید بخوایم یک تابع رو به یک تابع دیگه ورودی بدیم، از تابعی یک تابع دیگه رو خروجی بگیریم، و امثال این کارها.

توی جاوا، برای این که بتونیم با توابع مثل object برخورد بکنیم، از lambda expressionها کمک می‌گیریم. توی این داک می‌خوایم ببینیم lambda expressionها چی هستن و چطور استفاده می‌شن.

فیلتر کردن داده

فرض کنید رییس، یک List از Personهای مختلف داره و می‌خواد این List رو فیلتر کنه. کلاس Person به شکل زیره:

```
public class Person {
    public String name;
    public int age;
    public double height;
    public double weight;

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", height=" + height +
                ", weight=" + weight +
                '}';
    }
}
```

حالا، فرض کنید شما می‌خواید یه متد بنویسید که بتونه List<Person> رییس ما رو بر اساس پارامترهای مختلف فیلتر کنه. رییس گاهی بر اساس سن آدم‌ها، گاهی بر اساس وزنشون، گاهی بر اساس اسمشون و گاهی هم قدشون اون‌ها رو فیلتر می‌کنه. چه کار می‌کنید؟

راه حل اول: برای هر کدوم یه متد بنویسیم

توی راه حل اولمون، برای هر فیلترِ متفاوت رییس، یه متد می‌نویسیم. مثلا فرض کنید رییس بخواد اشخاصِ بالای ۱۸ سال رو از توی لیستش پیدا کنه، پس متد زیر رو براش می‌نویسیم:

```
void printEveryoneOver18(List<Person> people) {
    for (Person p : people) {
        if (p.age > 18) {
            System.out.println(p);
        }
    }
}
```

اگر رییس بخواد همهٔ آدم‌هایی که قدشون زیر ۱.۹۰ متره رو پیدا کنه، براش متد زیر رو می‌نویسیم:

```
void printEveryoneUnder190(List<Person> people) {
    for (Person p : people) {
        if (p.height < 190) {
            System.out.println(p);
        }
```

```
    }
}
```

یا اگر رییس بخواد هر کسی که اسمش با A شروع بشه رو پیدا کنه، می‌تونه از متد زیر استفاده کنه:

```
void printEveryoneWhoseNameStartsWithA(List<Person> people) {
    for (Person p : people) {
        if (p.name.startsWith("A")) {
            System.out.println(p);
        }
    }
}
```

می‌تونیم همین‌طور برای رییس متدهای جدید درست کنیم، ولی کد این متدها اونقدر با هم تفاوتی نداره. همهٔ اون‌ها کدشون به شکل زیره:

```
void printEveryoneWithSomeCondition(List<Person> people) {
    for (Person p : people) {
        if (/* check some condition about p */) {
            System.out.println(p);
        }
    }
}
```

و ما، برنامه‌نویس چندان خوبی نیستیم اگر اجازه بدیم کدمون انقدر تکراری باشه! بیاین سراغ راه حل بعدی بریم.

راه حل دوم: اینترفیس CheckPerson

بیاین یه اینترفیس جدید، به اسم CheckPerson ایجاد کنیم. این اینترفیس به شکل زیره:

```
public interface CheckPerson {
    boolean check(Person p);
}
```

اینترفیس ساده‌ایه. فقط یه متد check داره که یه شرط خاص رو در رابطه با Person چک می‌کنه. حالا می‌تونیم متدهای تکراری‌مون رو با متد زیر جایگزین کنیم:

```
void printEveryoneWithSomeCondition(List<Person> people, CheckPerson cp) {
    for (Person p : people) {
        if (cp.check(p)) {
            System.out.println(p);
        }
    }
}
```

حالا اگر رییس بخواد افراد بالای ۱۸ سال رو ببینه، کافیه یه کلاس کوچیک درست کنیم که CheckPerson رو پیاده‌سازی می‌کنه:

```
class CheckOver18 implements CheckPerson {
    @Override
    public boolean check(Person p) {
        return p.age > 18;
    }
}
```

و متد جدیدمون رو با یک instance از این کلاس صدا کنیم:

```
printEveryoneWithSomeCondition(people, new CheckOver18());
```

متد جدیدمون، خیلی متد قدرتمندتریه و انعطاف بیشتری داره، ولی حجم کدمون هنوز هم کم نشده! برای هر فیلتر جدید باید یک کلاس جدید ایجاد کنیم و به خاطر همین، حجم کدمون باز هم خیلی زیاده.[1]

راه حل سوم: Lambda Expressionها

اینترفیس CheckPerson، یک functional interfaceئه. به اینترفیس‌هایی که فقط یک متد دارن، functional interface می‌گن. به خاطر این که این اینترفیس‌ها فقط یک متد دارن، ما می‌تونیم به جای این که یک کلاس جدید برای پیاده‌سازی اون‌ها بسازیم، از Lambda Expressionها استفاده کنیم.

فرض کنید ما می‌خوایم تمام افراد بالای ۱۸ سال رو چاپ کنیم. می‌تونیم کلاس CheckOver18 رو دور بریزیم و متد printEveryoneWithSomeCondition رو به شکل زیر صدا بزنیم:

```
printEveryoneWithSomeCondition(people, (p) -> {
    return p.age > 18;
});
```

تیکهٔ جدید این کد، بخش زیره:

```
(p) -> {
    return p.age > 18;
}
```

این تکه از کد، شبیه یه پیاده‌سازی برای متد check از اینترفیس CheckPersonئه. توی این متد، ورودی pئه و خروجی یک booleanئه که نشون می‌ده سن p از ۱۸ بزرگتره یا نه. از اون‌جایی که

[1] به همچین کدی، اصطلاحا boilerplate code می‌گن؛ کدهای تکراری‌ای که برای کارهای خیلی ساده نوشته می‌شن.

CheckPerson تنها یک متد داره، خود جاوا می‌فهمه که این lambda expression پیاده‌سازی متد checkئه.

وقتایی که lambdaمون فقط یک ورودی داره، لازم نیست دور اون پرانتز بذاریم. کد بالا رو می‌شد به شکل زیر هم نوشت:

```
printEveryoneWithSomeCondition(people, p -> {
    return p.age > 18;
});
```

علاوه بر این، وقتی lambdaمون فقط یک returnئه، لازم نیست دور کدمون {} بذاریم و عبارت return رو بنویسیم. کد زیر هم کاملا درسته:

```
printEveryoneWithSomeCondition(people, p -> p.age > 18);
```

کلا lambda expression رو عموما به شکل بالا می‌نویسن.[2]

راه حل چهارم: اینترفیس‌های استاندارد جاوا

حتی این حد از کوتاه بودن کد هم برای ما کافی نیست! ما می‌تونیم از شر اینترفیس CheckPerson هم خلاص شیم. خود جاوا، یک اینترفیس جنریک به اسم Predicate داره که به شکل زیره:

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

}
```

پس ما می‌تونیم به جای CheckPerson، توی متدمون Predicate<Person> ورودی بگیریم:

```
void printEveryoneWithSomeCondition(List<Person> people,
                                    Predicate<Person> pp) {
    for (Person p : people) {
        if (pp.test(p)) {
            System.out.println(p);
        }
    }
}
```

جاوا از این functional interfaceها زیاد داره. بیاین دوتا دیگه از اون‌ها رو با هم بررسی کنیم:

[2] برای این که با انواع و اقسام شیوه‌های lambda نویسی آشنا بشین، این داک Oracle رو بخونین.

اینترفیس Consumer

این اینترفیس، به شکل زیره:

```
public interface Consumer<T> {
    void accept(T t);

}
```

همون‌طور که می‌بینید، فقط یک متد void داره که یک آبجکت T ورودی می‌گیره. به عنوان مثال، متد زیر رو در نظر بگیرین:

```
void doSomethingWithEveryone(List<Person> people,
                             Consumer<Person> c) {
    for (Person p : people) {
        c.accept(p);
    }
}
```

این متد، یک consumer ورودی می‌گیره و اون رو با تمام Personهای لیست ورودی‌ش اجرا می‌کنه. اگر این متد رو به شکل زیر صدا بزنیم باعث می‌شیم که تمام اعضای لیست people یکی یکی چاپ بشن:

```
doSomethingWithEveryone(people, p -> {
    System.out.println(p);
});
```

اینترفیس Function

اینترفیس Function، به شکل زیره:

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

}
```

این اینترفیس، برای زمانی به کار می‌ره که بخوایم توی lambdaمون چیزی خروجی بدیم. مثلا فرض کنید که یک لیست عدد داشته باشیم، و بخوایم یک عملیات ریاضی روی تمام اعضای اون انجام بدیم. می‌تونیم از متد زیر استفاده کنیم:

```
void numbersAfterOperation(List<Double> numbers,
                           Function<Double, Double> operation) {
    for (Double d : numbers) {
        System.out.println(operation.apply(d));
    }
}
```

حالا، می‌تونیم برای این که دو برابر هر عدد رو چاپ کنیم، این متد رو به شکل زیر صدا بزنیم:

```
numbersAfterOperation(numbers, n -> 2 * n);
```

یا اگر می‌خوایم رادیکال هر عدد رو چاپ کنیم، این متد رو به شکل زیر صدا می‌زنیم:

```
numbersAfterOperation(numbers, n -> Math.sqrt(n));
```

چیزی که یاد گرفتیم

توی این داک، ما با lambda expressionها آشنا شدیم. ما یاد گرفتیم که:

- functional interfaceها چی هستن و چطور می‌شه به کمک lambda expressionها، یک instance از اون‌ها تعریف کرد.
- functional interfaceهای استاندارد جاوا چی هستن.

منابع بیشتر

اگر دوست دارین بیشتر راجع به lambda expressionها یاد بگیرین، به داک رسمی Oracle یه سر بزنید.