

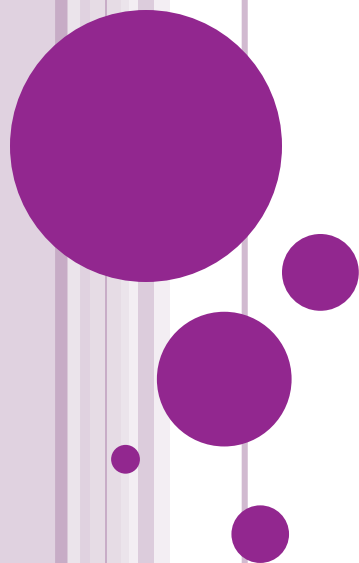
انجمن جاوا کاپ تقدیم می کند

دوره برنامه نویسی جاوا

# امکانات جدید جاوا در نسخه ۸

## JAVA 8 FEATURES

صادق علی اکبری



- کلیه حقوق این اثر متعلق به انجمن جاواکاپ است
- بازنشر یا تدریس آن چه توسط جاواکاپ و به صورت عمومی منتشر شده است، با ذکر مرجع (جاواکاپ) بلامانع است
- اگر این اثر توسط جاواکاپ به صورت عمومی منتشر نشده است و به صورت اختصاصی در اختیار شما یا شرکت شما قرار گرفته، بازنشر آن مجاز نیست
- تغییر محتوای این اثر بدون اطلاع و تأیید انجمن جاواکاپ مجاز نیست



# سرفصل مطالب



- در جاوا ۸ چه اتفاقاتی افتاده است؟
- چرا جاوا ۸ نسخه مهمی است؟

• عبارت لامبدا (Lambda Expression)

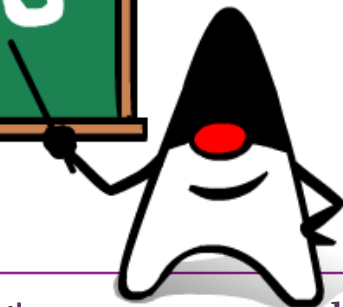
• برنامه‌نویسی تابعی (Functional Programming)

• واسط تابعی (Functional Interface)

• جویبار (Stream)

• جویبارهای موازی (Parallel Streams)

• امکانات جدید و گسترده کتابخانه‌ی جاوا ۸



# در جاوا ۸ چه اتفاقاتی افتاده است؟



- معرفی عبارت‌های لامبدا
- ارجاع به متد
- متغیرهایی که به متدها اشاره می‌کنند
- برنامه‌نویسی با رویکرد تابعی ممکن شده است
- برنامه‌هایی که کوتاه‌تر و گویاتر هستند
- معرفی مفهوم جویبار و جویبار موازی
- برای پردازش دنباله‌ای از داده‌ها
- امکانات بسیار گسترده‌ای با کمک مفاهیم فوق ایجاد شده است
- که باعث تسهیل برنامه‌نویسی می‌شوند



# چرا جاوا ۸ نسخه مهمی است؟

- جاوا ۸ گسترده‌ترین تغییر در تاریخ «زبان جاوا» است
  - حتی گسترده‌تر از جاوا ۵
  - که ساختارهای مهمی مانند Generic و Annotation را معرفی کرد
- برنامه‌نویس به جای چگونگی انجام کار، می‌تواند فقط هدف کار را توصیف کند
  - “what to do” instead of “how to do”



# چرا جاوا ۸ نسخه مهمی است؟ (ادامه)

- برنامه‌نویس جاوا ۸، می‌تواند «تابعی» بیاندیشد
- Functional Programming
- Thinking Functional
- این تغییر، در عمر جاوا بی‌سابقه است
- برنامه‌نویس جاوا عادت کرده که شیء‌گرا فکر کند
- جاوا ۸ کتابخانه و API زبان را گسترش داده و تقویت کرده است
- نیاز به کتابخانه‌های کمکی (مثل Apache Commons) کمتر می‌شود
- با معرفی جاوا ۸، دستخط برنامه‌نویسی جاوا به مرور تغییر خواهد کرد
- اگر دانش جاوا ۸ نداشته باشیم، بسیاری از کدها را نخواهیم فهمید





مروری بر مفاهیم جدید در جاوا ۸

An Overview of Java 8 Features

# متدهای پیش فرض برای واسطها

- یک واسط (interface):
- همانند کلاسی است که همه متدهای آن انتزاعی (abstract) هستند
- از جاوا ۸ به بعد، یک واسط می تواند متدهای غیرانتزاعی داشته باشد
- به این متدها، متد پیش فرض (Default Method) گفته می شود.

مثال: `interface Person {`

```
Date getBirthDate();
```

```
default Integer age(){
```

```
    long diff = new Date().getTime()-getBirthDate().getTime();
```

```
    return (int) (diff / (1000L*60*60*24*365));
```

```
}
```

```
}
```





# ارث‌بری از متد پیش‌فرض

- تعریف متد پیش‌فرض در کلاس‌هایی که واسط را پیاده‌سازی می‌کنند، اجباری نیست

```
class Student implements Person {
    private Date birthDate;
    public Date getBirthDate() {
        return birthDate;
    }
    @Override
    public Integer age() {
        long yearMiliSeconds = 1000L * 60 * 60 * 24 * 365;
        long currentYear = new Date().getTime() / yearMiliSeconds;
        long birthYear = getBirthDate().getTime() / yearMiliSeconds;
        return (int) (currentYear - birthYear);
    }
}
```

```
class Student implements Person {
    private Date birthDate;
    public Date getBirthDate() {return birthDate;}
}
```



# واسط تابعی (Functional Interface)

- واسطی که دقیقاً یک متد انتزاعی (abstract) دارد
- اگر بیش از یک متد دارد، یا تعدادی متد را به ارث گرفته است:
- همه این متدها، به جز یکی، باید تعریف پیش فرض داشته باشند



```
interface A {}
```

```
@FunctionalInterface  
interface B {  
    int f();  
}
```



```
interface C {  
    int f();  
    int g();  
}
```



```
interface D extends B {  
    int g();  
}
```



```
@FunctionalInterface  
interface E extends B {  
    default double g(){ return 2;}  
}
```



بالای تعریف یک واسط تابعی، می توانیم `@FunctionalInterface` را ذکر کنیم





# عبارت لامبدا (Lambda Expression)

مثال:

$r \rightarrow r * 2 * 3.14$

$(x, y) \rightarrow x + y$

• لامبدا یا لاندای ( $\lambda$ )

• یک عبارت لامبدا:

زیرا هر واسط تابعی، فقط یک متد نامشخص (انتزاعی) دارد

• قطعه کدی است که بدنه یک تابع را توصیف می کند

• و به عنوان یک واسط تابعی قابل استفاده است

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

یک عبارت لامبدا

```
Comparator<Person> comp =
```

```
(a, b) -> a.age().compareTo(b.age());
```



# مثال: کاربرد عبارتهای لامبدا

```
List<Person> people =
```

```
Arrays.asList(  
new Student("Ali", 1993),  
new Student("Taghi", 1990),  
new Student("Naghi", 1995));
```

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

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

اشاره: یک عبارت لامبدا به یک کلاس داخلی بی نام ترجمه نمی شود



# ارجاع به متد (Method Reference)

- امکانی جدید در جاوا ۸ که مانند اشاره گر به متد عمل می کند
- از :: برای ارجاع به متد استفاده می شود

```
class Str {  
    Character startsWith(String s) {  
        return s.charAt(0);  
    }  
}
```

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

ارجاع به متد

```
Str str = new Str();  
Converter<String, Character> conv = str::startsWith;  
Character converted = conv.convert("Java");
```

- هر جا که یک واسط تابعی مورد نیاز باشد:
- می توانیم از ارجاع به متد (و یا یک عبارت لامبدا) استفاده کنیم



# مثال برای ارجاع به متد

● ارجاع به متد

```
Converter<String, Character> conv = str::startsWith;
```

● ارجاع به متد استاتیک

```
Converter<String, Integer> converter = Integer::valueOf;
```

● ارجاع به سازنده (Constructor)

```
interface Factory<T> {  
    T create();  
}
```

```
Factory<Car> factory1 = Car::new;  
Car car1 = factory1.create();
```





واسط‌های تابعی جدید در جاوا ۸

Java 8 Functional Interfaces

# واسط‌های تابعی تعریف‌شده در جاوا ۸

- واسط‌های تابعی مختلف و مفیدی در جاوا ۸ (JDK 1.8) ایجاد شده
- بسیاری از واسط‌های مهم و معروف قبلی، واسط تابعی شده‌اند
  - مانند `Runnable` و `Comparator`
- واسط‌های تابعی جدیدی هم معرفی شده‌اند
  - مانند: `Consumer` و `Supplier`، `Function`، `Predicate`
  - در بسته `java.util.function` قرار دارند
  - مثلاً `java.util.function.Predicate`





# مثال برای Comparator

- در جاوا ۸ متدهای پیش فرض جدیدی به این واسط اضافه شده است

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

```
Person p1 = new Person("Ali");  
Person p2 = new Person("Taghi");
```

```
comparator.compare(p1, p2);           // < 0  
comparator.reversed().compare(p1, p2); // > 0
```

# Predicate

- یک واسط تابعی است: یک پارامتر می‌گیرد و **boolean** برمی‌گرداند
- این واسط، متدهای پیش‌فرض مختلفی دارد
- مانند `test` برای ارزیابی و `and` ، `or` و `negate` برای ترکیب `Predicate` ها

```
String st = "ok";  
boolean b;  
Predicate<String> notEmpty = (x) -> x.length() > 0;  
b = notEmpty.test(st); // true  
b = notEmpty.negate().test(st); // false
```

```
Predicate<String> notNull = x -> x != null;  
b = notNull.and(notEmpty).test(st); // true
```

```
Predicate<String> isEmpty = String::isEmpty;  
Predicate<String> isNotEmpty = isEmpty.negate();
```

# Function

- تابعی است که یک پارامتر می‌گیرد و یک خروجی تولید می‌کند
- با متد `apply` این تابع اجرا می‌شود
- متدهای پیش‌فرضی مانند `andThen` برای ترکیب زنجیروار تابع‌ها

```
Function<String, Integer> toInteger =  
    Integer::valueOf;
```

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

```
backToString.apply("123"); // "123"
```



# Supplier

---

- تابعی که یک شیء تولید (تأمین) می کند
- (برخلاف Function) هیچ پارامتری نمی گیرد
- اجرای تابع: با کمک متد **get**

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



# Consumer

- تابعی که فقط یک پارامتر می‌گیرد و خروجی ندارد
- یک شیء (پارامتر) را مصرف می‌کند
- اجرای تابع (مصرف شیء) با متد `accept` انجام می‌شود

```
Consumer<Person> greeter =  
    p -> System.out.println(p.getFirstName());  
  
greeter.accept(new Person("Ali"));
```



# مروری بر واسط‌های تابعی

واسط تابعی	توضیح
<code>Predicate&lt;T&gt;</code>	یک پارامتر از جنس <code>T</code> می‌گیرد و <code>boolean</code> برمی‌گرداند
<code>Consumer&lt;T&gt;</code>	یک پارامتر از جنس <code>T</code> می‌گیرد و آن را پردازش می‌کند ولی چیزی برنمی‌گرداند (یک شیء از جنس <code>T</code> را مصرف می‌کند)
<code>Supplier&lt;T&gt;</code>	یک شیء از جنس <code>T</code> تولید می‌کند (پارامتر نمی‌گیرد)
<code>Function&lt;T, U&gt;</code>	نوع <code>T</code> را به نوع <code>U</code> تبدیل می‌کند. پارامتری از جنس <code>T</code> می‌گیرد، آن را پردازش می‌کند و یک شیء از جنس <code>U</code> برمی‌گرداند.
<code>BiFunction&lt;T,U,V&gt;</code>	دو پارامتر به ترتیب از جنس <code>T</code> و <code>U</code> می‌گیرد و شیء از نوع <code>V</code> برمی‌گرداند
<code>UnaryOperator&lt;T&gt;</code>	یک عملگر تکی که یک شیء از جنس <code>T</code> می‌گیرد و شیء از همین جنس برمی‌گرداند
<code>BinaryOperator&lt;T&gt;</code>	دو پارامتر از جنس <code>T</code> می‌گیرد و شیء از همین جنس برمی‌گرداند



# مروری بر واسط‌های تابعی

واسط تابعی	توضیح
Predicate<T>	به همین ترتیب، واسط‌های تابعی دیگری هم تعریف شده‌اند:
Consumer<T>	BiConsumer <T , U>
Supplier<T>	BiPredicate <T , U>
	...
Function<T, U>	نوع T را به نوع U تبدیل می‌کند. پارامتری از جنس T می‌گیرد، آن را پردازش می‌کند و یک شیء از جنس U برمی‌گرداند.
BiFunction<T,U,V>	دو پارامتر به ترتیب از جنس T و U می‌گیرد و شیء‌ای از نوع V برمی‌گرداند
UnaryOperator<T>	یک عملگر تکی که یک شیء از جنس T می‌گیرد و شیء‌ای از همین جنس برمی‌گرداند
BinaryOperator<T>	دو پارامتر از جنس T می‌گیرد و شیء‌ای از همین جنس برمی‌گرداند



# چند مثال از واسط‌های تابعی

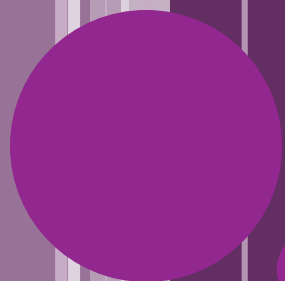
```
Map<Integer,String> map = ...
BiConsumer<Integer,String> biConsumer =
    (key,value) ->
System.out.println("Key:" + key + " Value:" + value);
map.forEach(biConsumer);
```

```
BiFunction<Integer, Integer, String> biFunction =
    (num1, num2) -> "Result:" + (num1 + num2);
System.out.println(biFunction.apply(20,25));
```

```
BiPredicate<Integer, String> condition =
    (i, s) -> i.toString().equals(s);
System.out.println(condition.test(10, "10")); //true
System.out.println(condition.test(30, "40")); //false
```







کوییز

• هر يك از اين عبارتهای لامبدا، معادل کدام واسطه تابعی قابل استفاده است؟

( ) -> **new** Person("Ali")

- Supplier<Person> s = () -> **new** Person("Ali");
- مثال: Supplier<Person> s = () -> **new** Person("Ali");

(p) -> System.**out.println**(p.*getName*())

- Consumer<Person>

(p) -> **new** Person(p.*getName*())

- UnaryOperator<Person>
- Function<Person, Person>

The slide features a dark purple background. On the left side, there are several vertical stripes of varying shades of purple and white. A cluster of five circles of different sizes, also in shades of purple, is positioned to the left of the main text.

# تمرین عملی

- تعریف واسط‌های تابعی

- پیاده‌سازی یک کلاس با وراثت از واسط تابعی

- استفاده از ۱- کلاس بی‌نام ۲- لامبدا ۳- ارجاع‌به‌متد برای اجرای

`Collections.sort`

- پیاده‌سازی `filterApples(applesList, predicate)`

- استفاده از لامبدا و ارجاع‌به‌متد به عنوان واسط‌های تابعی مختلف





جويبار  
STREAM



# مفهوم جویبار (Stream)

- جویبار یک دنباله از اشیاء است

- بر روی اعضای این دنباله، یک یا چند عمل انجام می شود

```
interface java.util.stream.Stream<T>
```

- بر روی هر `collection` می توانیم یک جویبار ایجاد کنیم

- برای پردازش اعضای این مجموعه با کمک جویبار

```
List<String> list = ...  
Stream<String> stream = list.stream();
```

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



# رفع سوءتفاهم درباره جویبار

- آن را با مفهوم جویبار در مباحث فایل و I/O اشتباه نگیرید
- هیچ ربطی به `InputStream` و `OutputStream` ندارد



# امکانات جویبار

- **جویبار** بر روی یک منبع (source) ایجاد می‌شود
- منبع، معمولاً یک collection است (مثلاً یک List یا Set)
- (ممکن است منبع یک جویبار چیز دیگری مانند آرایه، یک کانال IO و یا یک تابع مولد باشد)
- **جویبار**، بر اعضای یک مجموعه از اشیاء پیمایش (عبور) می‌کند
- و یک یا چند عمل (operation) بر روی این اعضا انجام می‌دهد
- دو نوع عمل بر روی جویبار ممکن است:



1. **عمل پایانی (terminal)**
  - داده‌ای از یک نوع خاص برمی‌گرداند
  - (مثلاً Integer, String, Void)
2. **عمل میانی (intermediate)**

○ همان جویبار را برمی‌گرداند: می‌توانیم عمل‌های دیگری را به زنجیره عمل‌ها اضافه کنیم





# نگاهی به نحوه کاربرد جویبار

- می‌خواهیم یک لیست از خودروها را پردازش کنیم
- یک جویبار بر روی این لیست ایجاد میکنیم: *stream()*
- از بین مواردی که رنگشان مشکی است
- یک فیلتر بر روی این جویبار ایجاد می‌کنیم که فقط مشکی‌ها را بپذیرد: *filter()*
- خودروها را براساس قیمت مرتب کنیم
- جویبار را مرتب می‌کنیم: *sorted()*
- دو مورد با کمترین قیمت را انتخاب کنیم
- اعضای مورد پردازش را محدود کنیم: *limit()*
- و اطلاعات این دو خودرو را چاپ کنیم
- هر یک از اعضای جویبار فوق را چاپ کنیم: *forEach()*

```
List<Car> list = ...
```

```
list
```

```
.stream()  
.filter(a->"Black".equals(a.color))  
.sorted((a,b)->a.price-b.price)  
.limit(2)  
.forEach(System.out::println);
```



# درباره جویبار

- با توجه به مثال قبل:

- به جای نحوه پردازش اطلاعات، فقط هدف (نتیجه) را توصیف می کنیم

- Program “**what to do**” instead of “**how to do**”

- مثال:

- `limit(2)`

- `forEach(System.out::println);`

- جاوا ۸ امکانات متنوعی را برای جویبارها فراهم کرده است

```
list.stream()
```

```
.parallel()
```

```
.filter((a)->a.price<40)
```

```
.forEach(System.out::println);
```

- حتی می توانیم فرایند اجرای جویبار را موازی کنیم

- (Multi-Thread)

- به سادگی و با فراخوانی یک متد: `parallel`



The left side of the slide features a decorative vertical band with several thin, parallel lines in shades of purple and white. To the right of these lines are five solid purple circles of varying sizes, arranged in a cluster that tapers towards the bottom.

درباره کلاس Optional

# مفهوم Optional

- یک کلاس جدید: تلاشی برای جلوگیری از `NullPointerException`

- هر `Optional` یک شیء در دل خود دارد

در ادامه، کاربردهای `Optional` را خواهیم دید

- خروجی (مقدار برگشتی) بسیاری از متدهای جدید، از جنس `Optional` است

- این متدها به جای `null`، یک `Optional` برمی گردانند که شیء درون آن `null` است

```
Optional<String> opt = Optional.of("salam");
boolean b = opt.isPresent(); // true
String s = opt.get(); // "salam"
String t = opt.orElse("bye"); // "salam"

// prints "s"
opt.ifPresent((s) -> System.out.println(s.charAt(0)));
```

# فلسفه Optional

- وجود Optional معجزه نمی کند!
- همچنان ممکن است NullPointerException ایجاد شود
- Optional کمک می کند که برنامه نویس حواسش را جمع کند
- وجود مقدار را (قبل از استفاده از آن) چک کند (مثلاً با متد `ifPresent`)
- اشتباه برنامه نویس و رخداد `NullPointerException` کمتر رخ می دهد
- کدهای تولیدشده هم تمیزتر و خواناتر و موجزتر می شوند

```
public Car findCar(String color) {...}
```

مثال:

```
System.out.println( findCar ("Black").getColor() );
```

```
public Optional<Car> findCar(String color) {...}
```

```
findCar("Black").  
ifPresent(car->System.out.println(car.getColor()));
```



The left side of the slide features a vertical stack of decorative elements. From top to bottom, there are several thin, light-colored vertical stripes, followed by a large, solid purple circle, a smaller solid purple circle, a medium-sized solid purple circle, a very small solid purple circle, and another medium-sized solid purple circle. The background of the entire slide is a solid, dark purple color.

**عمليات جويبار**

**Stream Operations**

# عملیات جویبار

- در ادامه، امکانات جویبارها را مرور می‌کنیم
- و عمل‌های مختلف (Stream Operations) بر روی یک جویبار را می‌بینیم
- ... و map ، filter ، forEach

در ادامه فرض می‌کنیم:

```
class Car{
    int price;
    String color;
    // getters & setters
    public Car(String color, int price) {
        this.color = color;
        this.price = price;
    }
    public String toString() {
        return "Car[color="+color+",price="+price+"]";
    }
}
```

```
List<Car> list =
    Arrays.asList(
        new Car("Black", 30),
        new Car("Black", 40),
        new Car("Black", 50),
        new Car("White", 20),
        new Car("Yellow", 60)
    );
```



# ForEach

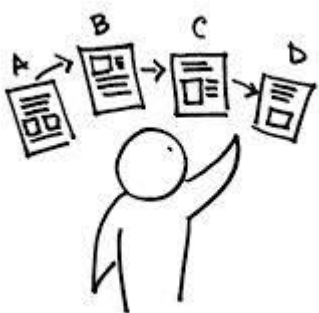
- یک عملیات را بر روی هر عضو جویبار انجام می‌دهد. مثال:

```
list.stream().forEach(System.out::println);
```

```
list.stream().forEach(a -> System.out.println(a.color));
```

- `forEach` یک عملیات پایانی (terminal) است یا میانی (intermediate)?

- پایانی. یعنی بعد از فراخوانی آن عمل دیگری بر روی جویبار قابل انجام نیست



- چه چیزی برمی‌گرداند؟

- هیچ (void)

- پارامترش از چه نوعی است؟

• **Consumer** (مثلاً `System.out::println`)

- مثل یک ارجاع به متد (مثال اول فوق) یا یک عبارت لامبدا (مثال دوم)





# Filter



- برخی از اعضای جویبار را حفظ می کند
- سایر اعضا که شرط موردنظر را ندارند نادیده گرفته می شوند

```
list.stream()  
.filter(a->"Green".equals(a.color));  
.forEach(System.out::println);
```

از اعضای لیست، آنهایی که

رنگشان سبز است

چاپ شوند

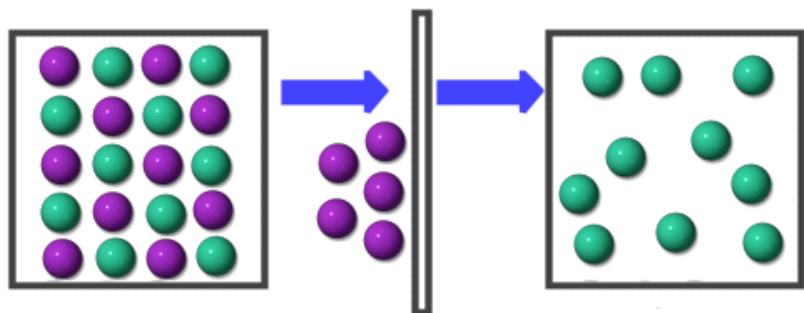
- یک عملیات پایانی است یا میانی؟

• میانی (intermediate)

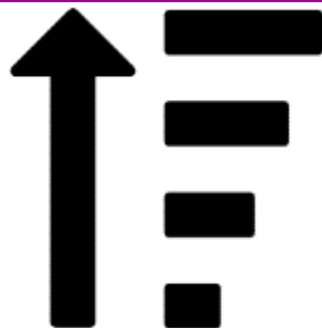
• یعنی همان جویبار را برمی گرداند

- پارامترش از چه نوعی است؟

• **Predicate** (شیء را بررسی می کند و `boolean` برمی گرداند)



# Sorted



- یک عمل میانی (intermediate operation)
- جویبار را مرتب (sort) می کند
- اگر اعضای جویبار Comparable باشند:

• عملیات sorted بدون پارامتر کار می کند `list.stream().sorted()`

- عملیات sorted امکان دریافت یک پارامتر از نوع Comparator را دارد
  - که نحوه مقایسه اعضای جویبار را مشخص می کند
  - مثل list که اعضای آن (Car) از جنس Comparable نیستند
  - یا در مواقعی که می خواهیم روش خاصی را برای مقایسه استفاده کنیم

`list.stream().sorted((a,b)->a.price-b.price)`

- نکته: عملیاتی مانند sorted و filter تغییری در منبع جویبار ایجاد نمی کنند
- مثلاً لیستی که جویبار بر روی آن ساخته شده، با فراخوانی sorted مرتب نمی شود



# Limit

- تعداد اعضای جویبار که پردازش می‌شوند را محدود می‌کند
- مثلاً `limit(2)` یعنی دو عضو ابتدای جویبار در نظر گرفته شوند
- عملیات موردنظر روی سایر اعضا انجام نمی‌شود
- نقطه مقابل `limit` ← `skip`
- `list.stream().limit(2).forEach(System.out::println);`
  - دو عضو ابتدای لیست را چاپ می‌کند
- `list.stream().skip(2).forEach(System.out::println);`
  - همه اعضای لیست به جز دو عضو اول را چاپ می‌کند



# Map

- یک عملیات میانی: هر یک از اعضای جویبار را به شیء دیگری تبدیل می کند

```
list.stream()  
.map(car->car.color)
```

- نحوه تبدیل را به صورت پارامتر دریافت می کند

- پارامترش یک Function است

- `java.util.function.Function<T, R>`

- جویباری از ماشین ها را به جویباری از رشته ها تبدیل می کند

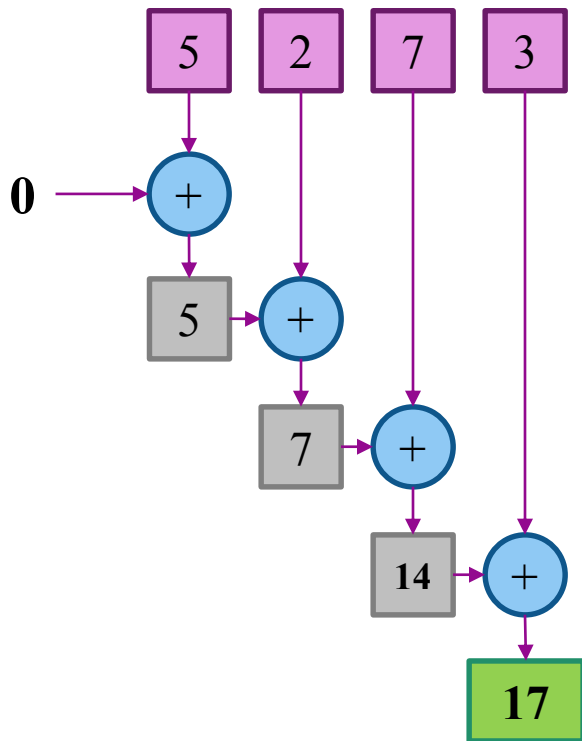
- `Stream<Car> → Stream<String>`

- نکته: این عملیات `map` ربطی به `java.util.Map` (مثلاً `HashMap`) ندارد

```
list.stream()  
.map(car->car.color)  
.filter(color ->  
color.startsWith("B"))  
.forEach(System.out::println);
```



# Reduce



- یک عمل پایانی
- یک مقدار تجمیعی از مجموعه اعضای جویبار استخراج می کند
  - مثال: از همه ماشین‌ها، مجموع قیمتشان را محاسبه کن
- پارامتر `reduce`: دو مقدار می‌گیرد و آن دو را ترکیب می‌کند
  - از این دو مقدار، یک مقدار حاصل می‌کند
- پارامتر `reduce` از چه نوعی است؟
  - BinaryOperator
- خروجی آن از جنس `Optional` است
  - چرا؟

```
Optional<Integer> sumOfPrices = list.stream()  
.map(car->car.price)  
.reduce((price1,price2)->price1+price2);
```

```
sumOfPrices.ifPresent(System.out::println);
```



# Count

- یک عملیات پایانی (terminal)

```
long lowPrices =  
list.stream()  
.filter((a) -> a.price < 40)  
.count();
```

- تعداد اعضای جویبار را برمی گرداند

- long برمی گرداند

- مثال:

- تعداد اعضای لیست که قیمتی کمتر از ۴۰ دارند



# Collect

- یک عملیات پایانی
- یک مجموعه از اشیاء (مثلاً یک List یا Set) از جویبار استخراج می کند
- یک Collector به عنوان پارامتر می گیرد
- مثال:

```
List<Car> newList = list.stream().filter((a) -> a.price<40)
.collect(Collectors.toList());
```

```
Set<Car> set = list.stream().filter((a) -> a.price<40)
.collect(Collectors.toSet());
```

Function<Car,String>

Function<Car,Car>

```
Map<String, Car> map = list.stream()
.collect(Collectors.toMap(car->car.color, car-> car));
```



# Match

---

```
boolean anyBlack =  
list.stream()  
.anyMatch(car -> car.color.equals("Black"));
```

```
boolean allBlack =  
list.stream()  
.allMatch(car -> car.color.equals("Black"));
```

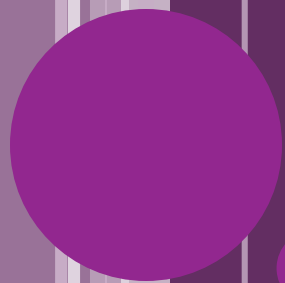
```
boolean noneBlack =  
list.stream()  
.noneMatch(car -> car.color.equals("Black"));
```





- `stream.distinct()`
- `Optional<T> stream.findAny()`
- `Optional<T> stream.findFirst()`
- `stream.max()`
- `stream.min()`
- `stream.toArray()`
- ...





کوییز

● قطعه برنامه‌ای بنویسید که:

- از مجموعه کارمندان شرکت (`List<Employee>`)
- از میان کسانی که متأهل هستند و حقوقشان کمتر از ۱۰۰۰ است
- نام ۱۰ نفر اول را به ترتیب حقوق (از کم به زیاد) چاپ کند

```
List<Employee> list = ...  
list.stream()  
.filter(e->e.isMarried)  
.filter(e->e.salary<1000)  
.sorted((a,b)->a.salary-b.salary)  
.limit(10)  
.forEach(e->System.out.println(e.name));
```



The slide features a dark purple background. On the left side, there are several vertical stripes of varying shades of purple and white. Below these stripes, there are five circles of different sizes, also in shades of purple, arranged in a descending pattern from top to bottom.

# تمرین عملی

# تمرین عملی برای جویبار

## ● تمرین map-reduce :

- فرض کنید یک مجموعه (set) از رشته‌ها داریم
- می‌خواهیم مجمول طول رشته‌ها را حساب کنیم
- ابتدا map و سپس reduce می‌کنیم
- راه ساده‌تری هم هست، ولی این الگو برای اجرای همروند مناسب است
- تغییرات بعدی:
- فقط رشته‌هایی پردازش شوند که با log شروع می‌شوند (filter)
- فقط ده رشته اول که طول بیشتری دارند پردازش شوند



The left side of the slide features a series of vertical stripes in various shades of purple and white. To the right of these stripes, there are several solid purple circles of varying sizes, arranged in a roughly vertical line that tapers towards the bottom.

# ساخت جویبار

# بازه‌های از اعداد

- واسطه‌هایی مانند `IntStream` و `LongStream` وجود دارند
- که می‌توانند بازه (`range`) اعداد ایجاد کنند
- با انواع اولیه کار می‌کنند
- `int` به جای `Integer` و `long` به جای `Long`

```
IntStream oneTo19 = IntStream.range(1, 20);  
IntStream oneTo20 = IntStream.rangeClosed(1, 20);
```

```
oneTo19.forEach(System.out::println);  
oneTo20.forEach(System.out::println);
```



# ایجاد دنباله‌ای از مقادیر با کمک جویبار

- با کمک امکان `iterate` می‌توانیم دنباله‌ای از مقادیر را توصیف کنیم

```
Stream.iterate(0, x -> x + 2)
    .limit(10)
    .forEach(System.out::println);
```

- پارامتر اول `iterate`: اولین عضو دنباله

- پارامتر دوم: یک `UnaryOperator`

- که نحوه ایجاد هر عضو بعدی از عضو قبلی را مشخص می‌کند

- نکته: متد `iterate` یک جویبار بینهایت می‌سازد

- با کمک `limit` تعداد اعضای دنباله که قرار است پردازش شوند، محدود شود





# روش‌های دیگر ساخت جویبار

- جویباری از مقادیر

```
Stream<String> stream = Stream.of("A", "B", "C");
```

- ساخت جویبار بر روی آرایه

```
String[] array = {"Ali", "Taghi"};
```

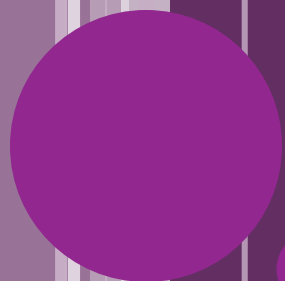
```
Arrays.stream(array);
```

- جویبار بر روی فایل

```
Stream<String> lines =
```

```
Files.lines(Paths.get("data.txt"));
```





کوییز

- این برنامه چه می کند؟

```
IntStream.range(2, 100)
    .filter(
        a -> IntStream.range(2, a - 1)
            .noneMatch(x -> a % x == 0)
    )
    .forEach(System.out::println);
```

- اعداد اول بازه دو تا ۱۰۰ را چاپ می کند





# جویبارهای موازی

# جویبارهای موازی (Parallel Streams)

- با کمک متد `parallel`

- عملیات مختلف به صورت موازی بر روی جویبار اجرا می‌شوند

```
list.stream().parallel()
```

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

- به این ترتیب: نیازی به ایجاد دستی `thread` ها نیست

- هم عملیات `sort` و هم `forEach` به صورت موازی انجام می‌شود  
(multi-thread)

- این که متد `parallel` در کجا فراخوانی شده مهم نیست



# جویبارهای موازی (ادامه)

- دنباله اعضا را به چند بخش تقسیم می کند
- و هر بخش در یک thread مجزا پردازش می شود
- به صورت پیش فرض، به تعداد هسته های پردازشی thread می سازد
- نکته:



- امکانات جدید جاوا از نسخه ۵ به بعد
- برای تسهیل و کم خطر شدن برنامه نویسی همروند
- Java 5: Thread Pools, Concurrent Collections
- Java 7: Fork/Join Framework



# دقت به نحوه استفاده از جویبارهای موازی

- استفاده از جویبار موازی، بسیار ساده و وسوسه‌برانگیز است
- اما استفاده نامناسب از جویبار موازی ممکن است:
  - موجب کاهش کارایی شود
  - نتایج اشتباه به بار آورد
- در استفاده از `parallel()` برای جویبارها باید دقت کنیم



# مثال: کاهش کارایی جویبار موازی

- کد زیر، مجموع اعداد یک تا  $n$  را به صورت موازی محاسبه می کند
- اگر `parallel` را فراخوانی نکنیم:

```
Stream.iterate(1L, i -> i + 1)  
    .limit(n)  
    .parallel()  
    .reduce((a, b) -> a + b);
```

• چندین برابر سریع تر می شود!

• فکر می کنید چرا؟

- زیرا `iterate` ماهیتی متوالی دارد

• تقسیم دنباله به چند بخش موازی، هزینه بر و کند است

• متد `iterate`، برای موازی سازی مناسب نیست

• (`parallel-friendly` نیست)






# تجزیه‌پذیری جویبار

- دیدیم موازی‌سازی `iterate` پرهزینه است
- زیرا تقسیم (تجزیه) اعضای این دنباله به چند بخش سخت است
- هرگاه تجزیه‌پذیری سخت باشد، موازی‌سازی کند می‌شود
- چند مثال دیگر

تجزیه‌پذیری عالی	تجزیه‌پذیری خوب	تجزیه‌پذیری ضعیف
ArrayList IntStream.range	HashSet TreeSet	LinkedList Stream.iterate

- کارایی کدام بیشتر است؟

- `LongStream.range(1, n+1).parallel()` 
- `Stream.iterate(1L, i -> i+1).limit(n).parallel()`



# اشتباه در نتایج با موازی سازی

```
class Accumulator {
    long total = 0;
    public void add(long value) {
        total += value;
    }
    long sideEffectParallelSum(long n) {
        Accumulator accumulator = new Accumulator();
        LongStream.rangeClosed(1, n)
            .parallel().forEach(accumulator::add);
        return accumulator.total;
    }
}
```

`sideEffectParallelSum(200_000)`

نتیجه باید 20\_100\_000 باشد  
ولی نتیجه عدد دیگری (مثل 19\_899\_923\_159) خواهد بود



The slide features a dark purple background. On the left side, there are several vertical stripes of varying shades of purple and white. A cluster of five circles of different sizes, also in shades of purple, is positioned to the left of the main text.

# تمرین عملی

# تمرین عملی برای جویبار

- استفاده از `parallel` و `foreach` به صورت همزمان
- محل فراخوانی `parallel` مهم نیست
- امکان فراخوانی `sequential()` در نهایت یکی اجرا می شود
- مثلاً: `stream.parallel().sequential()` ← `sequential`



The left side of the page features a series of vertical stripes in various shades of purple and white. To the right of these stripes, there are several solid purple circles of varying sizes, arranged in a cluster. The text 'جمع بندی' is positioned to the right of these circles.

جمع بندی

# اشاره گذرا به چند نکته مهم

- تا زمانی که یک عمل پایانی (terminal) فراخوانی نشود، هیچ یک از عملیات میانی (intermediate) اجرا نمی‌شود

- محدوده لامبدا: در یک عبارت لامبدا از متغیرهای محلی ( و سایر متغیرهای در دسترس) می‌توان استفاده کرد

- اما با این متغیرها باید مثل ثابت (final) برخورد کرد

```
int plus = 2;  
Stream.iterate(0, x -> x + plus)
```

- امکان تغییرشان وجود ندارد

- وراثت چندگانه در جاوا ۸ ممکن شده است

- با کمک وراثت از چند واسط که متدهای پیش فرض دارند

- مشکلات وراثت چندگانه چیست؟ آیا این وضعیت خطرناک است؟

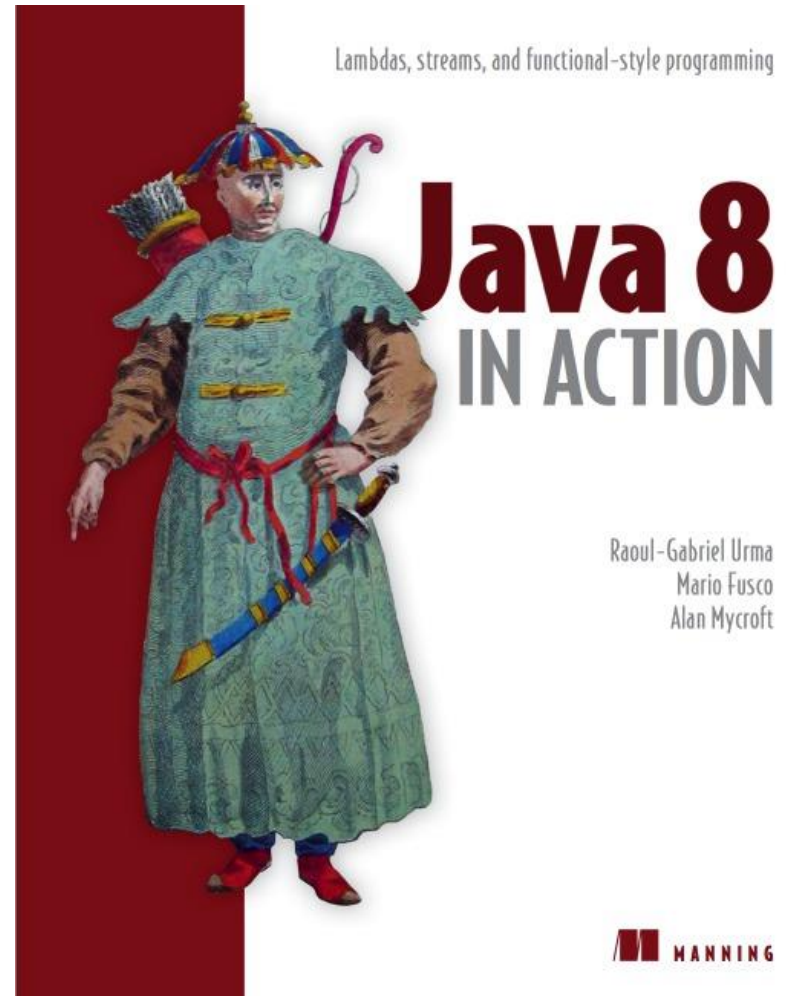
○ خیر.



- اهمیت و جایگاه نسخه ۸ جاوا
- عبارتهای لامبدا
- ارجاع به متد
- به عنوان شهروند درجه یک
- جویبار و جویبار موازی



- **Java 8 in Action:**  
Lambdas, Streams, and  
functional-style  
programming







## پایان (بخش اول)

بخش بعدی: نگاهی عمیق‌تر به مفاهیم و امکانات جاوا ۸