# کارگاه برنامه‌نویسی پیشرفته

## دستور کار شماره نه

## اهداف

آشنایی با ریسمان‌ها[1] در جاوا

کار با Executor Service

[1] Threads

# فهرست مطالب

# ریسمان‌ها[1] در جاوا

همان‌طور که می‌دانید در پروژه‌های بزرگ، وظایف[2] متعددی باید به طور هم‌زمان اجرا شوند. یکی از راه‌حل‌های این چالش، چند ریسمانی[3] است. در زبان جاوا، ریسمان به معنای یک واحد از برنامه است که به طور مجزا از واحدهای دیگر، وظایف خود را انجام می‌دهد.
برنامه‌هایی که تا به حال به زبان جاوا نوشته‌اید، همگی در یک ریسمان واحد به نام main اجرا می‌شدند. برای ساخت ریسمان‌های دیگر در جاوا راه‌های مختلفی وجود دارد، که به بررسی هر یک می‌پردازیم:

۱. کلاسی ساخته و در آن کلاس Thread را extend می‌کنیم:

```
public class MyThread extends Thread
```

حال، وظایف این ریسمان را در متد run که از کلاس والد بازنویسی[4] شده است، تعریف می‌کنیم:

```
@Override
public void run() {
   // declare some tasks here
}
```

یک نمونه[5] از کلاس بالا بسازید. برای اجرا کردن این ریسمان، متد start را صدا می‌زنیم:

```
MyThread myThread = new MyThread();
myThread.start();
```

**نکته:** اگر به طور مستقیم متد run را صدا بزنیم، تمام عملیات، داخل main thread انجام می‌شود. ولی با واسطه قرار دادن متد start، به این مشکل بر نخورده و یک ریسمان جدید ساخته می‌شود.
۲. کلاسی که اینترفیس[6] Runnable را پیاده‌سازی کرده، تعریف می‌کنیم و متد run را اورراید می‌کنیم:

```
public class MyRunnable implements Runnable {

   @Override
```

[1] Threads
[2] Task
[3] multi-threading
[4] override
[5] instance
[6] interface

```
    public void run() {
        // declare some tasks here
    }

}
```

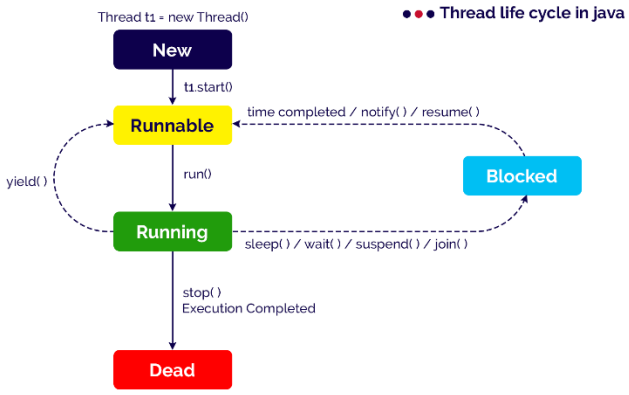
حال، یک شیء [1] از کلاس فوق ساخته و به کمک آن، یک ریسمان می‌سازیم تا با صدا زدن متد start، دستورات متد run در یک ریسمان جدید اجرا شوند.

```
Thread myThread = new Thread(new MyRunnable());
myThread.start();
```

توجه داشته باشید که: به دلیل اینکه با استفاده از این روش به مشکل لوزی [2] دیگر برنمیخوریم استفاده از این روش معمول‌تر است.

## چرخه‌ی زندگی ریسمان‌ها

همان‌طور که متوجه شده‌اید، هر ریسمان طول عمری دارد و در این مدت می‌تواند با حالت‌های متعددی روبه‌رو شود:




در ادامه، با وضعیت blocked آشنا خواهید شد.

[1] object
[2] Diamond Problem

# آشنایی با ExecutorService

مدیریت ریسمان‌ها[1]، از جمله start/terminate کردن هر یک، می‌تواند چالش بزرگی در روند نرم افزار ما باشد. یکی از راه‌حل‌های جاوا برای حل این مشکل، استفاده از اینترفیس[2] ExecutorService است. این اینترفیس، تعدادی تسک به صورت Runnable، دریافت کرده و آن‌ها را در مجموعه‌ای به نام استخر ریسمان[3] نگهداری می‌کند تا هر یک را به مطمئن‌ترین روش اجرا کند.
کلاس Executors برای ساخت تردپول، راه‌ها و فکتوری متدهای[4] مختلفی ارائه می‌دهد. برای مثال:

```
ExecutorService executorService = Executors.newCachedThreadPool();
/* Creates a thread pool that creates new threads as needed, but will reuse
previously constructed threads when they are available. */
```

```
ExecutorService executorService = Executors.newFixedThreadPool(10);
/* Creates a thread pool that reuses a fixed number of threads (10 threads
in this example) */
```

## نحوه استفاده از ExecutorService

یک Runnable یا کلاس برای اجرای تسک‌ها ساخته و در آن Runnable را implement کنید (توجه داشته باشید که extend کردن Thread در این حالت ممکن نیست):

```
public class MyTask implements Runnable
```

حال، چند شیء از این کلاس ساخته تا ExecutorService آن‌ها را اجرا کند:

```
MyTask task1 = new MyTask();
MyTask task2 = new MyTask();
MyTask task3 = new MyTask();

ExecutorService executorService = Executors.newCachedThreadPool();

executorService.execute(task1);
```

[1] Thread
[2] interface
[3] Thread pool
[4] Factory method

```
executorService.execute(task2);
executorService.execute(task3);
```

## فواید ExecutorService و تردپول[1]

- در ساختن ریسمان برای اجرای Runnableها، کارایی و سرعت برنامه بالاتر می‌رود.
- کنترل تمام تسک‌هایی که موازی پیش می‌روند، آسان‌تر خواهد بود.

در این‌صورت، برنامه‌نویس تسک‌های برنامه را مشخص کرده و مکانیزم اجرا شدن آن‌ها توسط ExecutorService تعیین می‌شود.

## مقایسه روش‌های ساخت thread

در این قسمت می‌خواهیم ۲ روش اصلی ساخت ریسمان را مقایسه کنیم:

۱. کلاسی که Runnable را implement کند، نماینده تسک‌هایی خواهد بود که نیاز به اجرا شدن در ریسمان مجزا دارند.

۲. بیشترین استفاده فعلی ما از ریسمان‌ها، متد run و محتویات آن است و در این مواقع (که نیازی به اوررایدکردن[2] دیگر متدهای کلاس Thread نیست)، استفاده از Runnable گزینه مناسب‌تری است. در این صورت می‌توان به روش‌های زیر از آن استفاده کرد:

۱. پاس دادن آن به یک نمونه[3] از کلاس Thread
۲. اجرا شدن توسط Executor هنگام استفاده از ExecutorService
۳. اجرا کردن تسک‌های آن در یک برنامه single threaded

۳. همانطور که می‌دانید، در جاوا ارث‌بری چندگانه[4] ممکن نیست. در نتیجه، اگر کلاس مورد نظرتان، کلاس دیگری را extend کند، دیگر قادر به extend کردن کلاس Thread نخواهید بود.

[1] Thread pool
[2] override
[3] instance
[4] Multiple inheritance

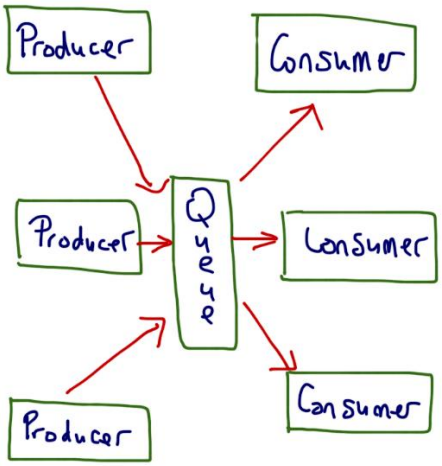
# Producer Consumer

ما تاکنون با برنامه‌نویسی چند ریسمانی[1] آشنا شدیم اما در این نوع برنامه‌نویسی مشکلی که امکان دارد رخ دهد درباره اطلاعات مشترک است که ریسمان‌ها برای عملکرد خود به پردازش آن‌ها نیازمند هستند. در ادامه به مثالی در این باره می‌پردازیم.

فرض کنید آقای یعقوب مسئول پخت پیتزا در سلف دانشگاه هست و پیتزاهای تولید شده را روی پیشخوان می‌گذارد تا Joey Tribbiani که دانشجوی مورد علاقه اوست، آن‌ها را بطور رایگان میل کند! فرض کنید این دو نفر از یک شمارنده استفاده کنند تا تعداد پیتزاهای باقی مانده را به هم اعلام کنند. حال اگر شمارنده به اشتباه دچار تغییر شود، امکان دارد که پیتزاها از ظرفیت پیشخوان بیشتر شوند و یا اینکه دانشجوی ما بخواهد پیتزا بخورد در حالیکه هیچ پیتزایی روی پیشخوان نباشد!

## مسئله producer-consumer

در این مسئله، تولیدکننده‌ها داده‌هایی را تولید می‌کنند و در یک صف قرار می‌دهند، سپس مصرف‌کنندگان آن‌ها را از صف گرفته و مصرف می‌کنند.




به عنوان مثال اداره‌ای را فرض کنید که در آن، پرونده‌های مختلف ابتدا بررسی می‌شوند تا مشکل آن‌ها پیدا شود و سپس کارمندان از دسته پرونده‌های بررسی شده، تعدادی از آن‌ها را برای حل مشکل انتخاب کرده و به هر یک رسیدگی می‌کنند. این کار باعث می‌شود تا بررسی پرونده‌ها به دو قسمت تقسیم شده و هر قسمت توسط افراد مختلف و نه به صورت تکی، انجام شود. در نتیجه، روند کاری سریع‌تر خواهد بود.

برای مطالعه بیشتر می‌توانید به لینک producer-consumer pattern مراجعه کنید.

امکان دارد یک داده به طور همزمان توسط چند مصرف کننده دچار تغییر شود و در نتایج عملیات اشکال به وجود آید؛ یا طبق مثال زده شده، یک پرونده به طور همزمان توسط چند کارمند بررسی شود که در این صورت امکان دارد چند نتیجه مختلف به وجود بیاید، که مطلوب ما نیست.

[1] Multi threading

## حل مشکل همزمانی

برای حل مشکل مطرح شده در قسمت قبل، جاوا راهکارهای مختلفی دارد که آن‌ها را باهم بررسی می‌کنیم:

- **Synchronized Blocks**

در این روش با استفاده از کلمه کلیدی synchronized و دادن یک شیء به آن، تمامی دستورات داخل بلوک‌های synchronized با شیء یکسان داده شده، تنها توسط یک ریسمان انجام می‌شود و تا وقتی که ریسمان وارد شده در این بلوک از آن خارج نشود، هیچ ریسمان دیگری وارد این بلوک نخواهد شد.

**نکته:** اگر کلمه this به بلوک synchronized پاس داده شود، تمامی بلوک‌های synchronized شده در این کلاس با کلمه کلیدی this تنها توسط یک ریسمان در یک زمان می‌تواند اجرا شود. حتی اگر بلوک‌ها در متدهای جداگانه باشند.

- **Synchronized Methods**

اگر در نوشتن یک متد از کلمه synchronized استفاده شود، تمامی کدهای داخل آن، در یک زمان تنها توسط یک ریسمان اجرا می‌شود و متد مربوطه تا پایان اجرا، قفل خواهد ماند.

- **Synchronized Wrappers**

تا کنون در جاوا با کالکشن‌هایی[1] آشنا شدید که امکان استفاده بطور همزمان توسط چند ریسمان را ندارند.
در جاوا تعدادی کالکشن[2] وجود دارد که در یک زمان تنها توسط یک ریسمان قابل استفاده هستند. ولی این کالکشن‌ها معایبی را به دنبال دارند؛ به عنوان مثال امکان استفاده از ایتریتور[3] را ندارند؛ زیرا ممکن است باعث ایجاد مشکل در حالت همزمانی شود.

```
Collection<Integer> syncCollection = Collections.synchronizedCollection(new ArrayList<>());
List<Integer> syncList = Collections.synchronizedList(new ArrayList<>());
Map<Integer, String> syncMap = Collections.synchronizedMap(new HashMap<>());
Map<Integer, String> syncSortedMap = Collections.synchronizedSortedMap(new TreeMap<>());
Set<Integer> syncSet = Collections.synchronizedSet(new HashSet<>());
SortedSet<Integer> syncSortedSet = Collections.synchronizedSortedSet(new TreeSet<>());
```

[1] collections
[2] collection
[3] iterator

- **پکیج java.util.concurrent**

در این پکیج نیز، تعدادی از کالکشن‌های مناسب برای همزمانی مانند:

- BlockingQueue
- LinkedBlockingQueue
- ConcurrentMap
- ConcurrentHashMap
- ConcurrentLinkedQueue
- CoppyOnWriteArraySet
- CopyOnWriteArrayList

وجود داشته که کاربردهای خاص خود را دارند.

برای مطالعه بیشتر می‌توانید به لینک **Thread Safe Collections** مراجعه کنید.

## انجام دهید: چند ریسمانی[1]

در این تمرین قصد داریم که مفاهیم چند ریسمانی که تا به حال یاد گرفته‌ایم را مرور کنیم و تسلط بیشتری به آن‌ها پیدا کنیم.

### راه اندازی پروژه

در این پروژه به شما ۲۰ فایل با فرمت txt. داده شده است که در هر یک، تعداد زیادی کلمه (**هر کدام در یک خط**) وجود دارد و برنامه باید تمام این فایل‌ها را خوانده و آن‌ها را آنالیز کند.

برای اینکه فایل‌ها را دانلود کنید باید این repository را clone کنید. فایل‌های مورد نیاز در پوشه Multi_threading_workshop_9 قرار دارد.

بهتر است که اول یک پروژه java بسازید و سپس این ۲۰ فایل را به داخل یک پوشه (مثلا به نام assets) انتقال دهید.

[1] multi thread

## فرایند کار نرم افزار

در این پروژه شما وظیفه دارید که هر کدام از فایل‌ها را **در یک ریسمان جدا و مخصوص به خود**، خط به خط بخوانید و آنها را آنالیز کنید. در نهایت، مشخصه‌های زیر را در مورد کل کلمات خوانده شده در تمام فایل‌ها را در کنسول[1] نشان دهید:

- تعداد کل کلمات (بدون تکرار)
- طولانی‌ترین کلمه و مقدار طول آن
- کوتاه‌ترین کلمه و مقدار طول آن
- میانگین طول تمام کلمات در تمام فایل‌ها

توجه داشته باشید که برنامه شما باید از نوع چند ریسمانی باشد.
برای نشان دادن مشخصه‌های بالا، می توانید یک منو طراحی کنید یا وقتی که محاسبات برنامه تمام شد، مولفه‌های بالا را با برچسب مربوط، در کنسول چاپ کنید.
نمونه‌ای از یک قالب یک منوی پیشنهادی:

```
Word Analyzer Project - AP Lab Workshop 9

1) Words Count
2) Longest word and its length
3) Shortest word and its length
4) All words length average
Please input the criteria number:
```

*۱قالب پیشنهادی برای منوی نرم افزار*

**توجه داشته باشید که:** قالب مطرح شده در عکس بالا صرفا یک پیشنهاد بوده و می‌توانید با استفاده از خلاقیت خود و به شرط اینکه تمام خروجی‌های مشخص شده در صورت سوال قابل دسترسی باشند، آن را تغییر دهید.
این تمرین را می‌توانید با ExecutorService و یا بدون آن انجام دهید. برای آشنایی بیشتر خودتان پیشنهاد می‌کنیم که هر دو روش را انجام دهید.

[1] console