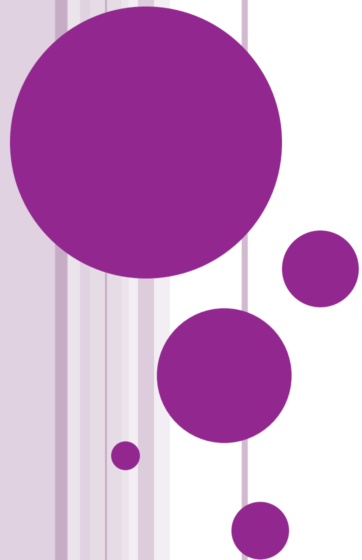
انجمن جاواکاپ تقدیم می‌کند

دوره برنامه‌نویسی جاوا

مفاهیم پیشرفته در برنامه‌های همروند
Advanced Concurrency



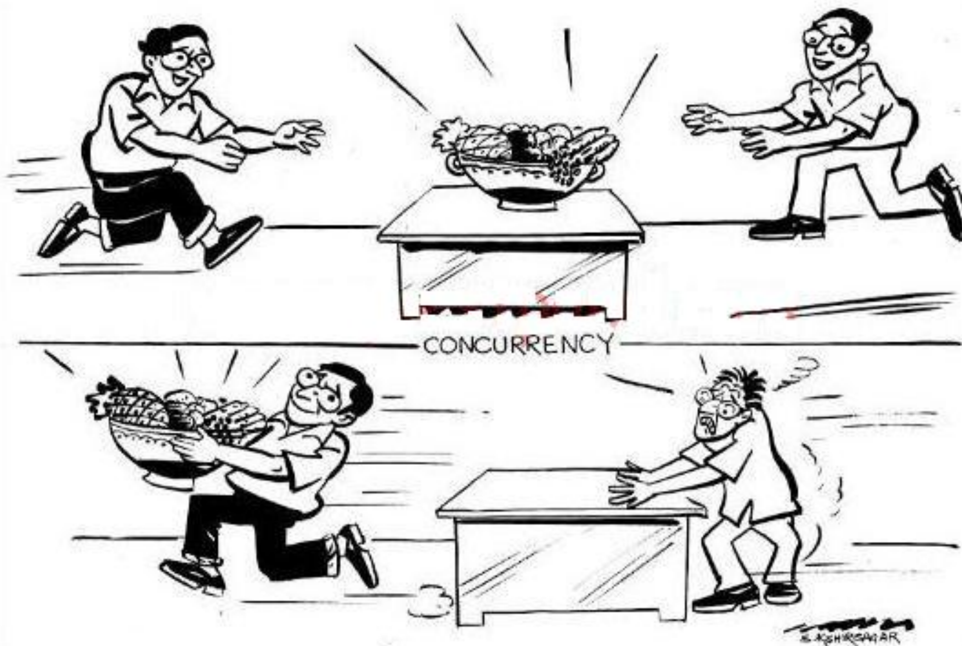
صادق علی‌اکبری

حقوق مؤلف

- کلیه حقوق این اثر متعلق به انجمن جاواکاپ است
- بازنشر یا تدریس آن‌چه 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

- امکانات جدید در جاوا در زمینه برنامه‌های همروند
- ظرف‌های همروند (concurrent collections)
- اشیاء هماهنگ‌کننده (synchronizers)
- کلاس‌های thread-safe
- اشیاء قفل (lock)
- متغیرهای اتمیک (atomic)
- اجراگر (executor)
- خزانه نخ (thread pool)
- مرور مفاهیم تئوری همروندی



مقدمه

مرور

- مفهوم نخ (Thread) و برنامه‌نویسی چندنخی (Multi-Thread)
- واسط Runnable و کلاس Thread
- نحوه ایجاد نخ جدید

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

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

- حالت‌های نخ‌ها
- مفهوم synchronization
- متدهای wait و notify

امکانات سطح بالا برای همروندی

- امکاناتی برای مدیریت دسترسی همزمان به اشیاء مشترک دیدیم:
  - synchronized, wait, notify, …
- این موارد، امکاناتی سطح پایین هستند
- از نسخه ۵ جاوا ، برخی امکانات سطح بالا و جدید برای مدیریت همروندی اضافه شد

- High-level concurrency APIs
  - این امکانات در بسته‌ی **java.util.concurrent** قرار دارند
  - معمولاً کارایی بهتری به نسبت امکانات قدیمی دارند
  - از پردازنده‌های چندهسته‌ای امروزی به خوبی بهره می‌برند
  - در بسیاری از کاربردها، برنامه‌نویسی را ساده‌تر می‌کنند
    - برای کاربردهای متنوع، امکانات متفاوت و متنوعی ایجاد شده است

کلاس‌های Thread-safe

مفهوم Thread-safe

- برخی از کلاس‌ها، thread-safe هستند: ایمن در همروندی
- استفاده از اشیاء این کلاس‌ها به صورت همروند، ایمن است
- از اشیاء این کلاس‌ها می‌توانیم به طور مشترک در چند نخ استفاده کنیم
- برای استفاده از این اشیاء در چند نخ همزمان، نیازی به قفل یا synchronized نیست
  - تمهیدات لازم در داخل همان کلاس پیاده‌سازی شده است
- مثال:

ایمن در همروندی (Thread-safe)	ناایمن در همروندی
Vector	ArrayList
ConcurrentHashMap	HashMap
StringBuffer	StringBuilder

چند نکته

- کلاس‌های معمولی بهترند یا معادل thread-safe آن‌ها؟
  - مثلاً بهتر نیست همیشه به جای ArrayList از Vector استفاده کنیم؟
  - خیر
  - اگر نیاز به استفاده مشترک از یک شیء در چند نخ نداریم، کلاس‌های معمولی کاراترند
  - تمهیداتی که برای thread-safety پیاده شده (مثل synchronized) اجرا را کندتر می‌کند
- اشیاء تغییرناپذیر (immutable) همواره thread-safe هستند
  - ویژگی‌های اشیاء تغییرناپذیر بعد از ساخت این اشیاء قابل تغییر نیست
  - مثلاً setter ندارند. مانند: String و Integer
  - امکان تغییر وضعیت آن‌ها وجود ندارد: استفاده از آن‌ها در چند نخ همزمان ایمن است

مثال

- کلاس‌های StringBuffer و StringBuilder هر دو برای نگهداری رشته هستند
  - برخلاف کلاس String، اشیاء این کلاس‌ها تغییرپذیر (mutable) هستند
  - مثلاً برای اضافه کردن یک مقدار به انتهای رشته، متد append دارند

```
StringBuffer buffer = new StringBuffer("12");
buffer.append("345");
String s = buffer.toString();
```

12345

  - با کلاس StringBuilder هم دقیقاً به همین شکل می‌توان کار کرد
- اما متدهای تغییردهنده در StringBuffer به صورت synchronized هستند

```
public synchronized StringBuffer append(String str) {…}
```

- StringBuffer یک کلاس thread-safe است، ولی StringBuilder نیست

ظرف‌های همروند (Concurrent Collections)

- راهی ساده برای این‌که یک کلاس thread-safe شود:
  همه متدها را synchronzied کنیم! (اما این راه کارایی مناسبی ندارد)
- از نسخه ۵ (JDK 1.5) بسته java.util.concurrent به جاوا اضافه شد
- این بسته شامل کلاس‌های جدید همروند است
- این کلاس‌ها، نه تنها thread-safe هستند،
  بلکه کارایی مناسبی در برنامه‌های همروند دارند
  - قفل‌ها به صورت بهینه گرفته و آزاد می‌شوند
- مانند:
  - ArrayBlockingQueue
  - ConcurrentHashMap
  - CopyOnWriteArrayList

مثلاً ConcurrentHashMap یک map است که به اشتراک گذاشتن اشیاء آن بین چند نخ، امن است

مثال: واسط BlockingQueue

- یکی از زیرواسط‌های Queue که thread-safe است
- معرفی متد put برای اضافه کردن و متد take برای حذف از صف
- هنگام استفاده از اشیائی از این نوع،

  در صورت لزوم هنگام خواندن و نوشتن، نخ در حال اجرا معطل می‌شود
  - اگر صف خالی باشد، هنگام خواندن متوقف می‌شود تا عضوی به صف اضافه شود
  - اگر ظرفیت صف پر باشد، هنگام نوشتن متوقف می‌شود تا عضوی از صف خارج شود
- مشابه مفهوم تولیدکننده/مصرف‌کننده (producer/consumer)
- ArrayBlockingQueue: پیاده‌سازی این واسط مبتنی بر آرایه با طول ثابت
- LinkedBlockingQueue: پیاده‌سازی مبتنی بر لیست پیوندی

اشیاء هماهنگ‌کننده

اشیاء هماهنگ‌کننده (Synchronizer)

- یک شیء هماهنگ‌کننده (synchronizer)
برای ایجاد هماهنگی بین چند نخ استفاده می‌شود
- چنین شیئی، یک وضعیت (حالت درونی) دارد
و با توجه به این وضعیت، به نخ‌های همکار، اجازه اجرا یا توقف می‌دهد
- کلاس‌های متفاوتی برای کاربردهای مختلف ایجاد شده است. مانند:

- Semaphore
- CountDownLatch
- Exchanger
- CyclicBarrier

در این زمینه، قبلاً امکانات سطح پایین‌تری
مانند wait و notify را دیده بودیم

سمافور (Semaphore)

- دسترسی به منابع مشترک را کنترل می‌کند
- یک عدد برای تعیین تعداد «استفاده‌کننده‌های همزمان» نگهداری می‌کند
  - این عدد حالت سمافور را مشخص می‌کند
    (حداکثر تعداد نخ‌هایی که همزمان از منبع مشترک استفاده می‌کنند)
- متدهای اصلی سمافور: acquire() و release()
- هر نخ قبل از استفاده از شیء مشترک، باید متد acquire را فراخوانی کند
  - اگر حالت سمافور صفر باشد، این متد بلاک می‌شود (اجرای نخ متوقف می‌شود)
    (تعداد نخ‌هایی که همزمان وارد شده‌اند، از عدد اولیه تعیین‌شده بیشتر شده است)
- هر نخ در پایان استفاده از شیء مشترک، باید متد release را فراخوانی کند
  - موجب آزاد شدن یک نخ (که منتظر acquire است) می‌شود

مثال: پیاده‌سازی تولیدکننده/مصرف‌کننده با سمافور

```java
synchronized(list){
  list.add(obj);
}
semaphore.release();
```

تولیدکننده

```java
semaphore.acquire();
synchronized (list) {
  obj = list.remove(0);
}
```

مصرف‌کننده

- اشیاء list و semaphore بین چند نخ به اشتراک گذاشته می‌شود
- شیء سمافور مشترک (sem) به صورت زیر ایجاد شده است:

```java
Semaphore sem = new Semaphore(0);
```

- دو دغدغه:
  - دو نخ، همزمان از لیست مشترک استفاده نکنند ← synchronized
  - اگر لیست خالی است، نخ مصرف‌کننده متوقف شود ← سمافور

CountDownLatch هماهنگ‌کننده

- یک هماهنگ‌کننده (Synchronizer) دیگر
- به چند نخ اجازه می‌دهد تا پایان یک شمارش معکوس متوقف شوند
- کاربرد: در نخ‌های مختلف تعداد مشخصی عملیات باید رخ دهند، تا امکان ادامه برخی نخ‌ها فراهم شود
- متدهای اصلی این کلاس :
  - متد await : منتظر پایان شمارش معکوس می‌شود
  - متد countDown : شمارش معکوس را یک واحد پیش می‌برد

```
CountDownLatch latch = new CountDownLatch(2);
```

Thread#1

```
latch.await();
System.out.println("Finished!");
```

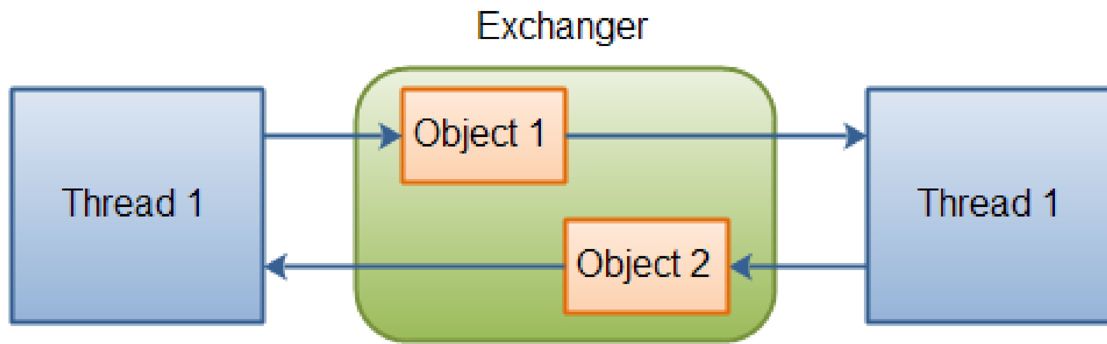
Thread#2

```
latch.countDown();
```

Thread#3

```
latch.countDown();
```

کلاس Exchanger



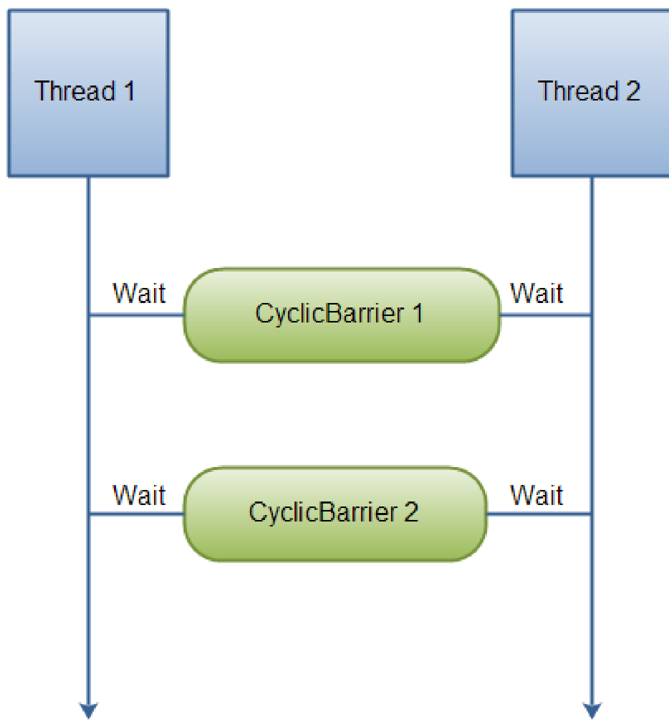
- هماهنگ‌کننده‌ای (synchronizer) دیگر: برای هماهنگی و تبادل داده بین دو نخ
- هر نخ متد exchange را فراخوانی می‌کند و یک پیغام (شیء) به آن پاس می‌کند
- نخی که exchange را صدا کرده متوقف می‌شود تا نخ دیگر هم آن را صدا کند
- در این لحظه هر دو نخ از توقف آزاد می‌شوند (ادامه اجرا)
- هر نخ پیغامی (شیء) که نخ دیگر برای آن ارسال کرده را دریافت می‌کند

```
Exchanger<String> e = new Exchanger<>();
```

```
e.exchange("x=2");
```
Thread#1

```
e.exchange("y=3");
```
Thread#2

هماهنگ‌کننده‌های دیگر



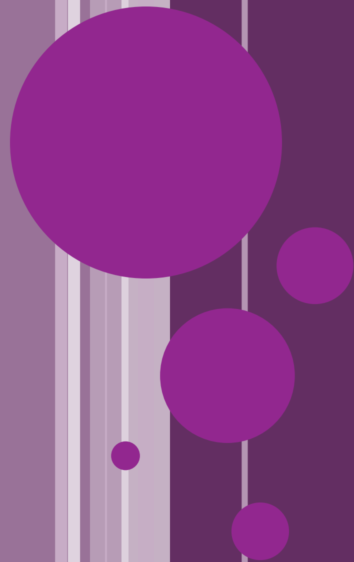

```java
CyclicBarrier barrier
        = new CyclicBarrier(3);
```

```java
barrier.await();
```

```java
barrier.await();
```

```java
barrier.await();
```

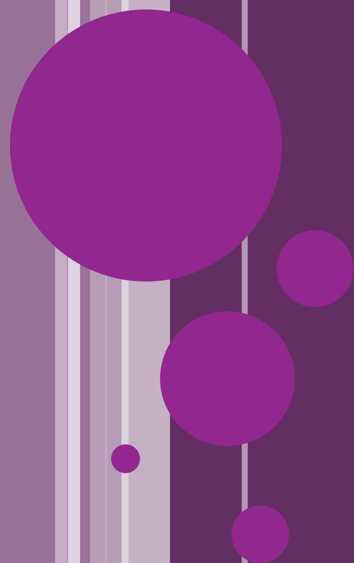
- کلاس CyclicBarrier
  - کاربرد: چند نخ، برای رسیدن یکدیگر به یک نقطه مانع مشترک صبر کنند
  - یعنی بعضی از نخ‌ها، هر یک در نقطه‌ای از اجرا، باید منتظر رسیدن بقیه باشند
- کلاس Phaser (از جاوای ۷)
  - مشابه CyclicBarrier و CountDownLatch اما امکانات منعطف‌تر



تمرین عملی

تمرین عملی

- مرور امکانات هماهنگ‌کننده‌ها
  - Sempahore و CountDownLatch
  - با یک مثال واقعی



کلاس‌های اتمیک (Atomic)

عملیات اتمیک (Atomic)

- گاهی یک عملیات ظاهراً ساده ممکن است به چند دستور سطح پایین ترجمه شود
  - مثلاً ++i به i=i+1 ترجمه شود (خواندن i ، عملیات جمع و تغییر مقدار i)
  - نباید درمیانه اجرای این عملیات، نخ دیگری از این متغیر استفاده کند
- گاهی چند عملیات ظاهراً مستقل، به یک دستور سطح پایین قابل ترجمه هستند
- **عملیات اتمیک**: همه عملیات، یک‌جا اجرا شود (در میان اجرای آن، نخ دیگری وارد نشود)
  - کل عملیات به صورت یک دستور سطح پایین اجرا شود (در صورت پشتیبانی پردازنده)
  - یا با گرفتن قفل پیاده‌سازی شود: برای نخ‌های دیگر به توقف (blocking) منجر شود
- گاهی برای انواع ساده (مثل عدد و آرایه) به عملیات اتمیک نیاز داریم
  - مانند افزایش یک متغیر عددی، و یا «خواندن و تغییر» مقدار یک متغیر

کلاس‌های اتمیک (Atomic Class)

- برخی عملیات بر روی اشیاء این کلاس‌ها به صورت اتمیک ممکن شده است
  - اجرای متغیر اتمیک کاراتر از پیاده‌سازی با قفل و synchronized و ... خواهد بود
- کلاس‌های اتمیک جاوا در بسته‌ی java.util.concurrent.atomic هستند
  - مانند AtomicInteger ، AtomicLong ، AtomicLongArray و ...
- متغیرهای کلاس‌های اتمیک
  - thread-safe : به صورت امن در چند نخ قابل اشتراک هستند
  - lock-free: برای استفاده در چند نخ نیازی به قفل و synchronized و ... ندارند

برخی کلاسهای اتمیک

- **AtomicBoolean**
- **AtomicInteger** (extends Number)
- AtomicIntegerArray
- **AtomicLong** (extends Number)
- AtomicLongArray
- AtomicReference
- AtomicReferenceArray

مثال: AtomicInteger

- برخی متدهای این کلاس:

```
AtomicInteger()
AtomicInteger(int initVal)
int get()
void set(int newVal)
int getAndSet(int newValue)
int getAndIncrement()
int getAndDecrement()
boolean compareAndSet (int expect, int update)
```

- مثال:

```
AtomicInteger at = new AtomicInteger(12);
int a_12 = at.get();
at.set(20);//at=20
int a_21 = at.incrementAndGet();//at=21
int b_21 = at.getAndIncrement();//at=22
int a_27 = at.addAndGet(5);//at=27
boolean is9_false = at.compareAndSet(9, 3);//at=27
boolean is27_true = at.compareAndSet(27, 30);//at=30
```

- متغیرهای اتمیک، thread-safe هستند
  - بدون نیاز به قفل و synchronized و ... از آن‌ها در چند نخ استفاده می‌کنیم

كوييز

سؤال

- از بین کلاس‌های زیر،

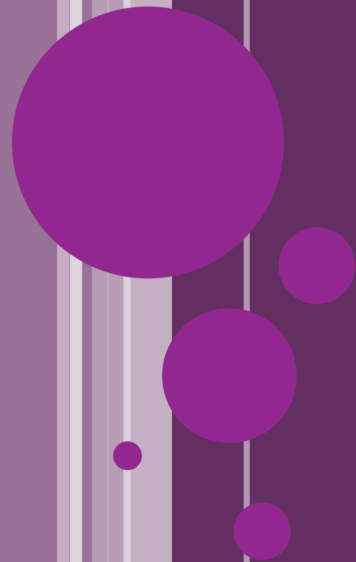
۱- چه کلاس‌هایی تغییرناپذیر (Immutable) هستند؟

۲- چه کلاس‌هایی thread-safe هستند؟

۳- اشیاء چه کلاس‌هایی را بدون نیاز به قفل و synchronized می‌توانیم بین چند نخ به اشتراک بگذاریم؟

- String ۱ ۲و۳
- Integer ۱ ۲و۳
- AtomicLong ۲و۳
- ArrayList
- HashMap
- ConcurrentHashMap ۲و۳

نکته:
- سؤال ۲ و ۳ یکی هستند
- هر چه immutable باشد، thread-safe هم هست (و نه لزوماً برعکس)



مفهوم Lock

واسط Lock

- بسته‌ی java.util.concurrent.locks از جاوا ۵ اضافه شد
  - شامل واسط‌ها و کلاس‌های جدید برای گرفتن قفل در برنامه‌های همروند
  - مثل واسط‌های Lock ، ReadWriteLock و Condition
- با کمک lock: دسترسی همزمان چند نخ به یک بخش حیاتی را محدود می‌کنیم
  - تا در هر لحظه حداکثر یکی از نخ‌ها در حال اجرای بخش حیاتی باشد
  - امکانی مشابه synchronized ، اما پیچیده‌تر و انعطاف‌پذیرتر
  - Synchronized ← قفل ضمنی و Lock ← قفل صریح
  - هدف هر دو ساختار یکی است:

در هر لحظه تنها یک نخ به منبع مشترک (بخش بحرانی) دسترسی دارد

اشیاء Lock

- با کمک شیء Lock ، محل گرفتن و آزادسازی قفل توسط برنامه‌نویس مشخص می‌شود
- متدهای مهم واسط Lock :
  - متدهای lock و unlock برای گرفتن و آزادسازی قفل (مثل بلوک synchronized)
  - متد lock قفل همان شیء Lock را می‌گیرد و unlock قفل را آزاد می‌کند
  - متد tryLock مانند lock عمل می‌کند، ولی اگر قفل آزاد نبود، متوقف نمی‌شود
    - non-blocking
    - اگر نتواند قفل را بگیرد: false برمی‌گرداند
- یکی از کلاس‌هایی که واسط Lock را پیاده‌سازی کرده است: **ReentrantLock**

مثال

```
Lock l = new ReentrantLock();
l.lock();
try {
  ... // critical section
}finally {
  l.unlock();
}
```

واسط ReadWriteLock

- یک بخش بحرانی از یک برنامه را در نظر بگیرید که در آن:
  - بسیاری از نخ‌ها متغیر مشترک را می‌خوانند
  - بعضی از نخ‌ها متغیر مشترک را تغییر می‌دهند
- اگر همروندی را با روش‌های معمولی مثل synchronized کنترل کنیم:
  - حتی اگر دو نخ بخواهند متغیر مشترک را بخوانند، یکی باید منتظر پایان دومی بماند
  - این وضعیت کارا نیست
- واسط ReadWriteLock دو قفل مجزا در نظر می‌گیرد:
  یکی برای نویسنده‌ها و یکی برای خواننده‌ها!
  - اجازه می‌دهد چند نخ که فقط می‌خواهند متغیر مشترک را بخوانند، همزمان اجرا شوند
  - متد readLock برای قفل خواندن و متد writeLock برای نوشتن

مثال: کلاس ReentrantReadWriteLock

- واسط ReadWriteLock را پیاده‌سازی کرده است. مثال:

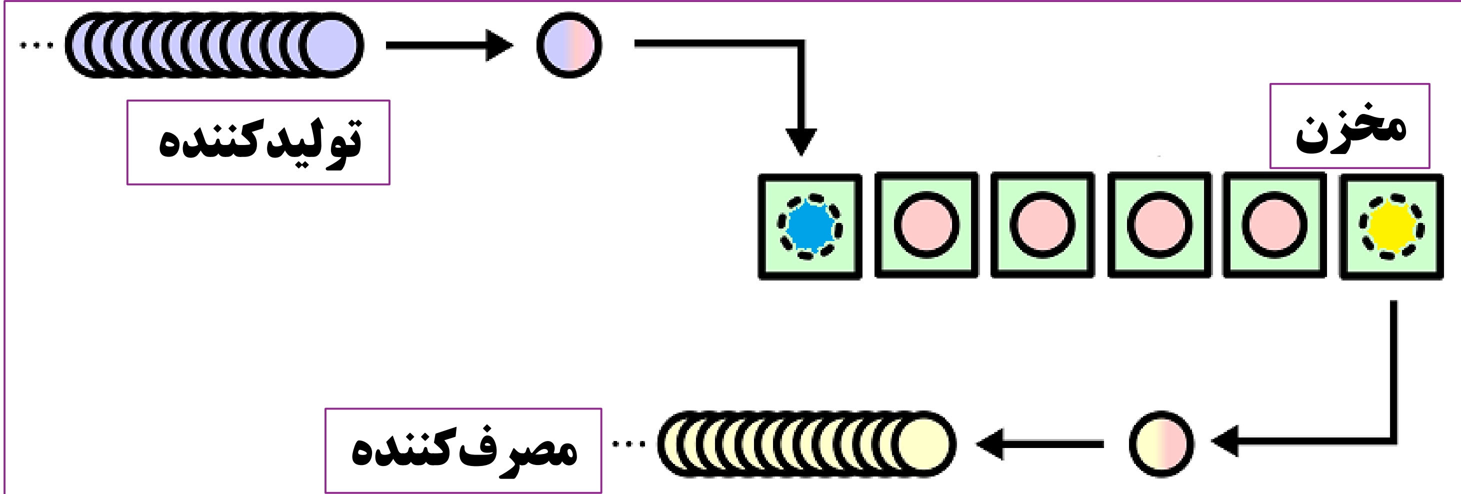
```
List<Double> list= new LinkedList<>();
ReadWriteLock lock = new ReentrantReadWriteLock();
```

```
class Reader extends Thread{
 public void run() {
  lock.readLock().lock();
  System.out.println(list.get(0));
  lock.readLock().unlock();
 }
}
```

```
class Writer extends Thread{
 public void run() {
  lock.writeLock().lock();
  list.add(0, Math.random());
  lock.writeLock().unlock();
 }
}
```

کوییز

سؤال



- مسأله تولیدکننده/مصرف‌کننده را به خاطر بیاورید
- آیا استفاده از ReadWriteLock برای این مسأله مناسب است؟
  - در نخ تولیدکننده با قفل writeLock کار کنیم (ابتدا قفل کنیم و در انتها آزاد کنیم)
  - در نخ مصرف‌کننده با قفل readLock کار کنیم
- پاسخ:
  - خیر. در این مسأله، مصرف‌کننده هم یک تغییردهنده است (فقط Reader نیست)
  - هم تولیدکننده و هم مصرف‌کننده مخزن را تغییر می‌دهند (هر دو Writer هستند)

چارچوب Executor

مدیریت ایجاد نخ

- برای ایجاد یک نخ اجرایی جدید، یک راه دیدیم:
  - ایجاد شیء از Thread و فراخوانی متد start

```
Thread t = new MyThread();
t.start();
```

```
Thread t = new Thread(runnable);
t.start();
```

- ولی این راه معمولاً مناسب نیست!
- برنامه‌های بزرگ معمولاً این‌گونه هستند:
  - در بخشی از برنامه، نیاز به اجرای فعالیتی مشخص (task) تعیین می‌شود
  - در بخشی مجزا از برنامه، نحوه ایجاد نخ‌ها، زمان‌بندی و ... مدیریت می‌شود

task submission

- برنامه‌نویسی که یک وظیفه را پیاده‌سازی می‌کند، نحوه مدیریت نخ‌ها را تعیین نمی‌کند

چارچوب‌های اجراگر نخ‌ها (Executors)

- فرایند ایجاد و اتمام یک نخ جدید پیچیده و پرهزینه است
- برای مدیریت این کار الگوریتم‌های مناسبی پیاده‌سازی شده است
- به اشیائی که ایجاد و اجرای نخ‌ها را مدیریت می‌کنند، اجراگر (Executor) می‌گویند

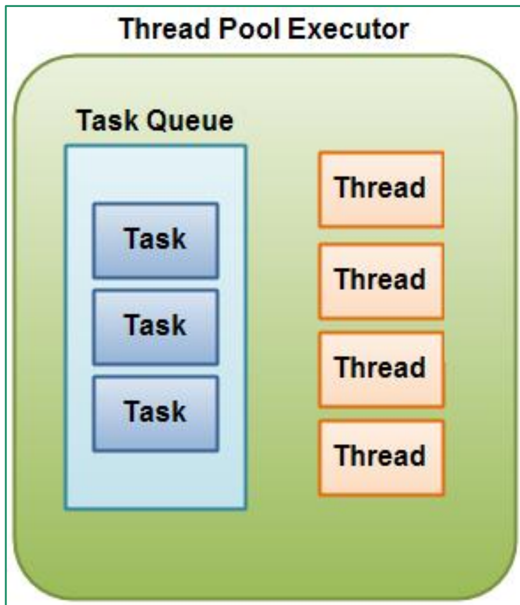
```
package java.util.concurrent;
interface Executor {
  void execute(Runnable command);
}
```

- واسط Executor:
  - متد execute یک شیء Runnable را در یک نخ اجرا می‌کند
    - شاید از یکی از نخ‌هایی که قبلاً ایجاد شده، استفاده کند
    - شاید یک نخ جدید برای اجرای آن ایجاد کند
    - شاید در همین نخ جاری اجرا کند

نحوه ایجاد و مدیریت نخ به نوع Executor بستگی دارد

خزانه نخ (Thread Pool)

- معمولاً به شیئی از نوع Executor برای مدیریت ایجاد و اجرای نخ‌ها نیاز داریم



- کلاس‌های مختلفی به عنوان Executor طراحی شده‌اند
- این اشیاء معمولاً یک خزانه نخ (thread pool) دارند
- خزانه نخ (thread pool) :
  - تعداد m کار به طور همزمان در n نخ اجرا می‌شوند
    - تعداد کارها ممکن است از تعداد نخ‌ها بیشتر شود
  - از ایجاد و خاتمه مکرر نخ‌ها (که حافظه و زمان را مصرف می‌کند) جلوگیری می‌شود
  - برای هر کار جدید، حتی‌الامکان یکی از نخ‌های بیکار thread pool به کار می‌رود
  - الگویی برای کاهش تعداد نخ‌های ایجادشده در برنامه

کلاس کمکی Executors

- یک کلاس کمکی در بسته java.util.concurrent
- متدهای استاتیک ساده‌ای برای برگرداندن شیء از انواع Executor دارد. مثال:
  - متد newSingleThreadExecutor : خزانه‌ای با یک نخ
    - کارها، پشت سر هم در همان یک نخ اجرا می‌شوند (برای آغاز کار۲،کار۱ باید تمام شود)
  - متد newFixedThreadPool: خزانه‌ای با تعداد مشخصی نخ
    - کارها، در تعداد مشخصی نخ اجرا می‌شوند
    - اگر تعداد کارها بیشتر از تعداد نخ‌ها باشد، اجرای برخی کارها بعد از اتمام کارهای قبلی
  - متد newCachedThreadPool: برای هر کار جدید یک نخ ایجاد می‌کند
    - با پایان کار یک نخ، آن را نگه می‌دارد و برای اجرای کارهای بعدی بازاستفاده می‌کند

مثال

```
Executor e = Executors.newFixedThreadPool(2);
Runnable runnable = new Runnable(){
 public void run() {
  for (int i = 0; i < 4; i++)
     System.out.println(Thread.currentThread().getId()+":"+i);
 }
};
for (int i = 0; i < 3; i++)
  e.execute(runnable);
```

```
9:0
10:0
10:1
10:2
9:1
10:3
9:2
10:0
9:3
10:1
10:2
10:3
```

شروع کار اول

شروع کار دوم

شروع کار سوم در یکی از نخ‌های قبلی

مثال

```
Executor e = Executors.newSingleThreadExecutor();
Runnable runnable = new Runnable(){
 public void run() {
  for (int i = 0; i < 4; i++)
     System.out.println(Thread.currentThread().getId()+":"+i);
 }
};
for (int i = 0; i < 3; i++)
  e.execute(runnable);
```

```
9:0
9:1
9:2
9:3
9:0
9:1
9:2
9:3
9:0
9:1
9:2
9:3
```

شروع کار اول

شروع کار دوم

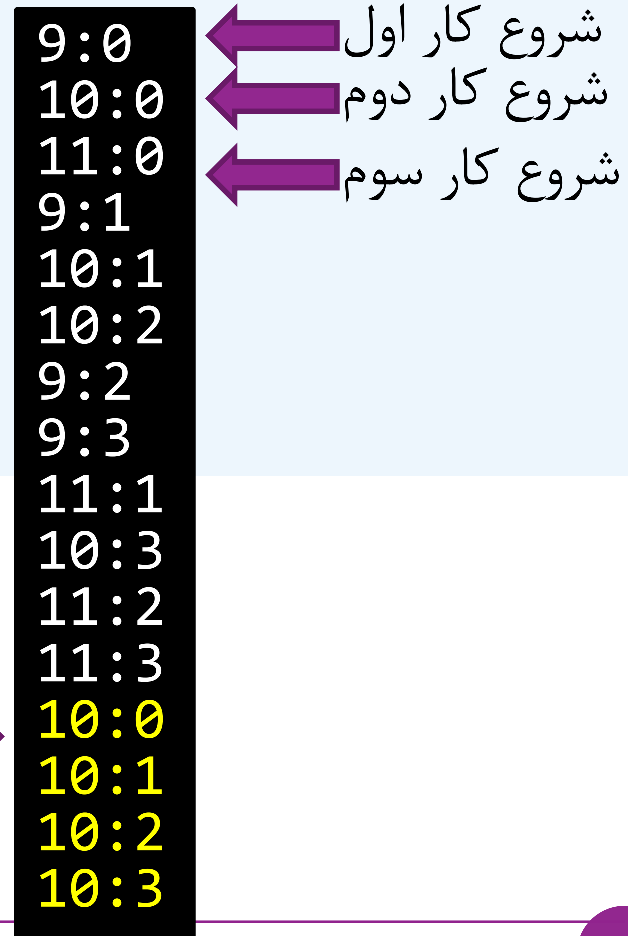
شروع کار سوم

همه کارها در یک نخ

مثال

```java
Executor e = Executors.newCachedThreadPool();
Runnable runnable = new Runnable(){
 public void run() {
  for (int i = 0; i < 4; i++)
    System.out.println(Thread.currentThread().getId()+":"+i);
 }
};
for (int i = 0; i < 3; i++)
  e.execute(runnable);

Thread.sleep(1000);
e.execute(runnable);
```



شروع کار اول

شروع کار دوم

شروع کار سوم

شروع کار بعدی در یکی از نخ‌های قبلی

واسط‌های Callable و Future

واسط java.util.concurrent.Callable

```
interface Callable<V> {
  V call() throws Exception;
}
```

- مشکل واسط Runnable:
  - متد run مقدار برگشتی ندارد (void است)
  - اما گاهی یک عملیات باید یک مقدار محاسبه یا تولید کند
  - با کمک run این خروجی باید در یک منبع مشترک نوشته شود
  - نخی که خروجی عملیات را لازم دارد، باید منتظر پایان آن نخ بماند
  - همروندی و دسترسی مشترک نخ‌ها هم باید کنترل شود
- واسط Callable: واسطی مشابه Runnable که یک عملیات را توصیف می‌کند
- به جای متد run ، متد call دارد
- برخلاف run مقداری برمی‌گرداند و ممکن است خطا پرتاب کند

واسط java.util.concurrent.Future

- این واسط خروجی یک عملیات را نشان می‌دهد (که احتمالاً در نخی دیگر اجرا شده)
- متدهایی دارد برای:
  - بررسی این که آیا عملیات تمام شده است (isCancelled و isDone)
  - انتظار برای پایان عملیات و دریافت نتیجه عملیات (get)
  - قطع عملیات (cancel)
- بسیاری از کلاس‌های موجود Executor ، متدی با نام submit دارند:

```
interface ExecutorService extends Executor {
    <T> Future<T> submit(Callable<T> task);
    ...
}
```

مثال

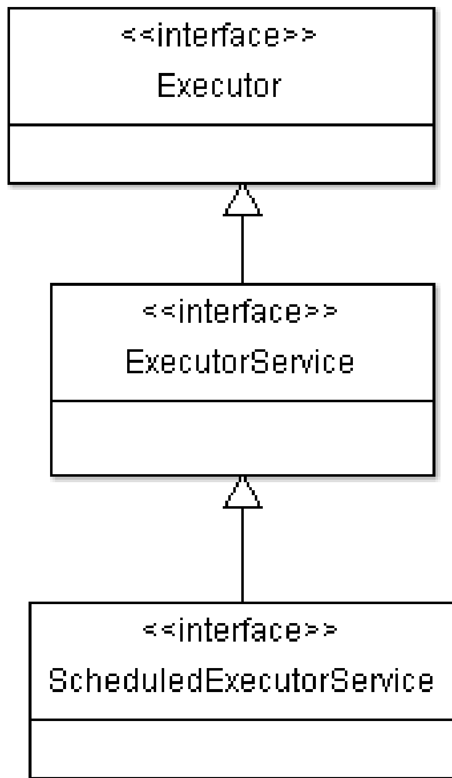
می‌خواهیم مجموع طول چند رشته
را به صورت چند نخی محاسبه کنیم

```
class WordLengthCallable implements Callable<Integer> {
 private String word;
 public WordLengthCallable(String word) {
  this.word = word;
 }
 public Integer call() {
  return word.length();
 }
}
```

```
ExecutorService pool = Executors.newCachedThreadPool();
Set<Future<Integer>> set = new HashSet<>();
String[] words = { "Ali", "Taghi", "Naghi" };
for (String word : words) {
  Callable<Integer> callable = new WordLengthCallable(word);
  Future<Integer> future = pool.submit(callable);
  set.add(future);
}
int sum = 0;
for (Future<Integer> future : set)
   sum += future.get();
System.out.println("The sum of lengths is " + sum);
```

13

مروری بر واسط‌های Executor



- واسط Executor
  - دارای متد `void execute(Runnable task)`
- واسط ExecutorService
  - دارای متد `<T> Future<T> submit(Callable<T> task)`
- واسط ScheduledExecutorService
  - دارای متد schedule برای زمان‌بندی اجرای یک کار با تأخیر:

```
public <V> ScheduledFuture<V> schedule(
Callable<V> callable, long delay, TimeUnit unit);
```

مفهوم ThreadLocal

- کلاس ThreadLocal:
  - اشیائی که از این کلاس ایجاد می‌شوند، فقط داخل همان نخ قابل استفاده خواهند بود
  - حتی اگر دو نخ، یک کد یکسان را اجرا کنند:
    این دو نخ نمی‌توانند متغیرهای ThreadLocal یکدیگر را بخوانند یا تغییر دهند
  - وقتی چنین متغیری new می‌شود، عملیات new در هر نخ فقط یک بار اجرا می‌شود
- متغیرهای ThreadLocal فقط در محدوده یک نخ استفاده می‌شوند
- کاربرد:
  - به‌اشتراک‌گذاری یک داده با بخشی از برنامه که در همین نخ اجرا خواهد شد
  - به این ترتیب نیازی به کنترل دسترسی همزمان به این متغیر (با قفل و ...) نیست

مثال

```java
class Task implements Runnable{
 ThreadLocal<Integer> tl = new ThreadLocal<>();
 public void run() {
   tl.set(tl.get() == null ? 1 : tl.get() + 1);
   long thrID = Thread.currentThread().getId();
   System.out.println(thrID + ":" + tl.get());
 }
}
```

```
9:1
9:2
9:3
9:4
9:5
```

```java
Executor e = {
   Executors.newSingleThreadExecutor();
   Executors.newFixedThreadPool(10);
}
Task task = new Task();
for (int i = 0; i < 5; i++)
   e.execute(task);
```

```
9:1
11:1
13:1
10:1
12:1
```

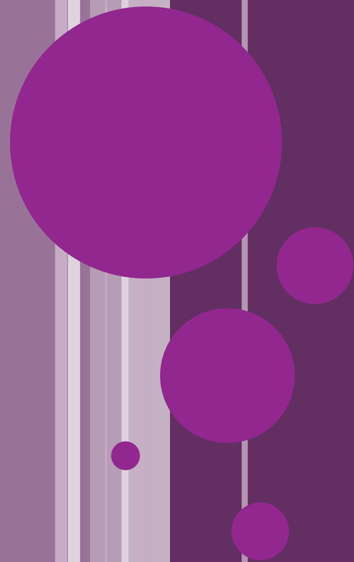
کوییز

کوییز: خروجی برنامه زیر؟

```
class SumTask implements Callable<Integer> {
   private int num = 0;
   public SumTask(int num) {
      this.num = num;
   }
   @Override
   public Integer call() throws Exception {
      int result = 0;
      for (int i = 1; i <= num; i++)
        result += i;
      return result;
   }
}
```

```
ExecutorService service =
              Executors.newSingleThreadExecutor();
SumTask sumTask = new SumTask(10);
Future<Integer> future = service.submit(sumTask);
Integer result = future.get();
System.out.println(result);
```

55



مرور مفاهیم تئوری همروندی

معضلات برنامه‌های همروند

- دسترسی و تغییر همزمان متغیر مشترک، یکی از معضلات برنامه‌های همروند است
  - شرایط مسابقه (Race condition)

    که درباره این معضل و راههای کنترل و جلوگیری از آن صحبت کردیم
- اما به واسطه استفاده نابجا یا ناکارامد از امکاناتی مثل lock و synchronized
  - نه تنها ممکن است کارایی و سرعت برنامه به شدت افت کند
  - بلکه ممکن است اشکالات دیگری ایجاد شود، مانند:

    گرسنگی (Starvation) و بن‌بست (Deadlock)

تشخیص امکان و جلوگیری از گرسنگی و بن‌بست در برنامه‌های بزرگ بسیار مشکل است

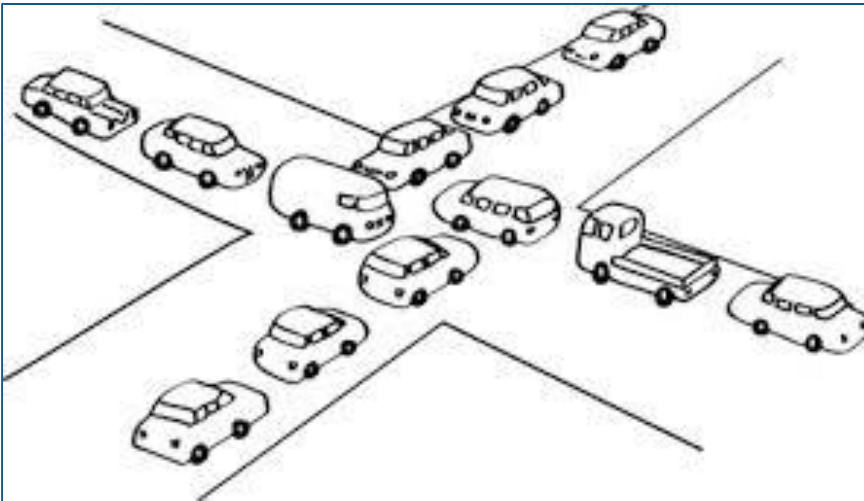
- طراحی یک برنامه همروند باید به‌گونه‌ای باشد که:

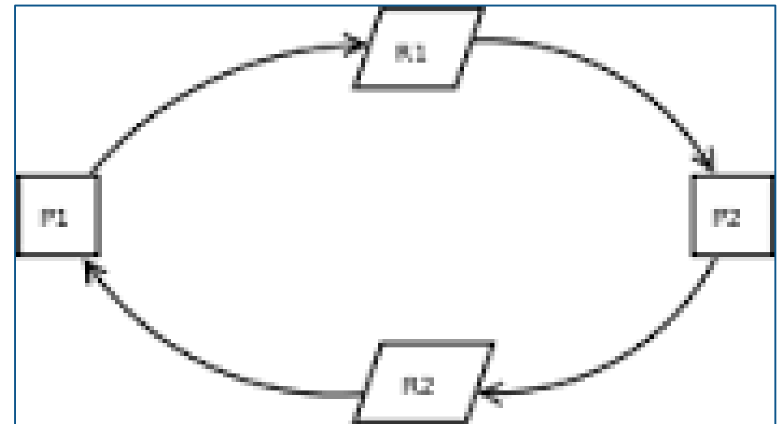
  ۱- امکان دسترسی نامناسب به منابع مشترک را ندهد

  ۲- کارایی مناسبی ارائه کند. به‌ویژه از گرسنگی و بن‌بست جلوگیری کند.

بن‌بست (deadlock)

- شرایطی که در آن چند نخ برای همیشه متوقف می‌مانند زیرا منتظر یکدیگرند
- چند بخش همروند وجود داشته باشد که هر یک منتظر پایان دیگری باشد
  - مثلاً نخ ۱ قفل الف را گرفته ولی برای ادامه اجرا منتظر آزاد شدن قفل ب است
    همزمان نخ ۲ قفل ب را گرفته و منتظر آزاد شدن قفل الف است





گرسنگی (starvation)

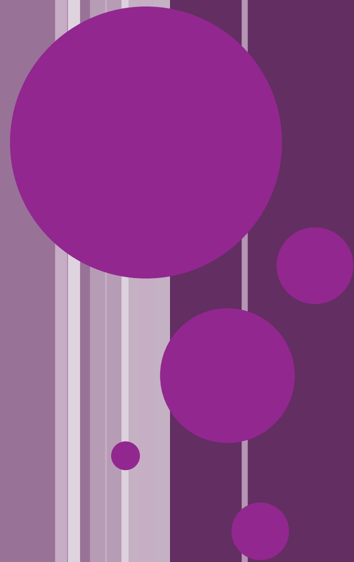
- برخی از نخ‌ها همواره منتظر باشند و هیچ‌وقت نوبت اجرای آن‌ها نشود
- مثلاً یک نخ همواره منتظر گرفتن قفل برای ورود به بخش بحرانی بماند
  - زیرا نخ‌های دیگری همواره زودتر قفل را می‌گیرند
- مسأله گرسنگی کمتر از بن‌بست به وجود می‌آید ولی کشف و رفع آن هم پیچیده‌تر است
  - معمولاً به خاطر الگوریتم‌های ساده (ناکارامد) زمان‌بندی و اولویت‌بندی ناشی می‌شود
  - سیستم‌های عامل جدید از الگوریتم‌های مناسبی برای زمان‌بندی نخ‌ها استفاده می‌کنند
  - یک برنامه به خاطر الگوریتم بدوی زمان‌بندی بین نخ‌ها ممکن است ایجاد گرسنگی کند
- مثال:

نخ‌های خجالتی: تا وقتی منبع مشترک قفل است ۱۰ ثانیه صبر کن، سپس یک کار یک‌دقیقه‌ای

نخ‌های بی‌پروا: تا وقتی منبع مشترک قفل است ۱۰ میلی‌ثانیه صبر کن، سپس یک کار یک‌ساعته

مرور چند مفهوم

- انحصار متقابل (Mutual Exclusion یا Mutex)
- انتظار مشغول (Busy Waiting)
  - انتظار در متدهایی که دیدیم (مثل wait ، sleep ، lock و ...) این گونه نیستند
- شرایط مسابقه (Race Condition)
- بخش بحرانی (Critical Section)
- منبع مشترک (Shared Resource)
- قفل (lock)
- مانیتور (Monitor)



تمرین عملی

تمرین

- مشاهده برنامه‌های
- ProducerConsumer1
- ProducerConsumer2
- ProducerConsumer3

جمع‌بندی

جمع‌بندی

- کلاس‌های thread-safe
- امکانات جدید در جاوا در زمینه برنامه‌های همروند
  - ظرف‌های همروند (concurrent collections)
  - اشیاء هماهنگ‌کننده (syncrhonizers)
  - اشیاء قفل (Lock)
  - متغیرهای اتمیک (atomic)
  - اجراگر (Executor)
  - خزانه نخ (thread pool)
  - واسط‌های Callable و Future
- مفهوم گرسنگی و بن‌بست

مطالعه کنید

- فصل ۲۳ کتاب دایتل Java How to Program (Deitel & Deitel)

23 Concurrency 957

- سایر منابع:

- https://docs.oracle.com/javase/tutorial/essential/concurrency/
- http://tutorials.jenkov.com/java-concurrency

تمرین

- مسأله تولیدکننده/مصرف‌کننده را با کمک سمافور در حالتی پیاده‌سازی کنید که اندازه بافر (مخزن) هم محدود باشد
  - چند نخ تولیدکننده و چند نخ مصرف‌کننده ایجاد کنید
- برنامه‌ای چندنخی بنویسید که بن‌بست ایجاد کند
- برنامه‌ای چندنخی بنویسید که گرسنگی ایجاد کند
- از کلاس HashMap در یک برنامه چندنخی استفاده کنید و نشان دهید که این کلاس thread-safe نیست
  - از ConcurrentHashMap استفاده کنید و نشان دهید مشکل برطرف می‌شود

جستجو کنید و بخوانید

- در جاوا ۸ چه امکانات مهم و مفیدی برای برنامه‌نویسی چندنخی اضافه شده است؟
  - Parallel Streams
- واسط Condition
- کلاس جدید ForkJoinPool (از جاوای ۷)
- متغیرهای volatile (کلیدواژه volatile)
- مسأله غذاخوردن فلاسفه (Dining Philosophers Problem)
- مفهوم livelock و تفاوت آن با گرسنگی و بن‌بست

پایان