

به نام خدا



برنامه‌سازی پیشرفته

دانشگاه شهید بهشتی . دانشکده مهندسی کامپیوتر

دکتر مجتبی وحیدی اصل

برنامه نویسی چند نخی (Multi Thread Programming)

آراس ولیزاده

فهرست مطالب

1. مالتی تردینگ در جاوا
2. مالتی تسکینگ
3. تعریف Thread در جاوا
4. کلاس Thread در جاوا
5. چرخه حیات یک Thread
6. روش‌های ایجاد Thread
7. زمانبند Thread
8. متدهای مهم در Thread

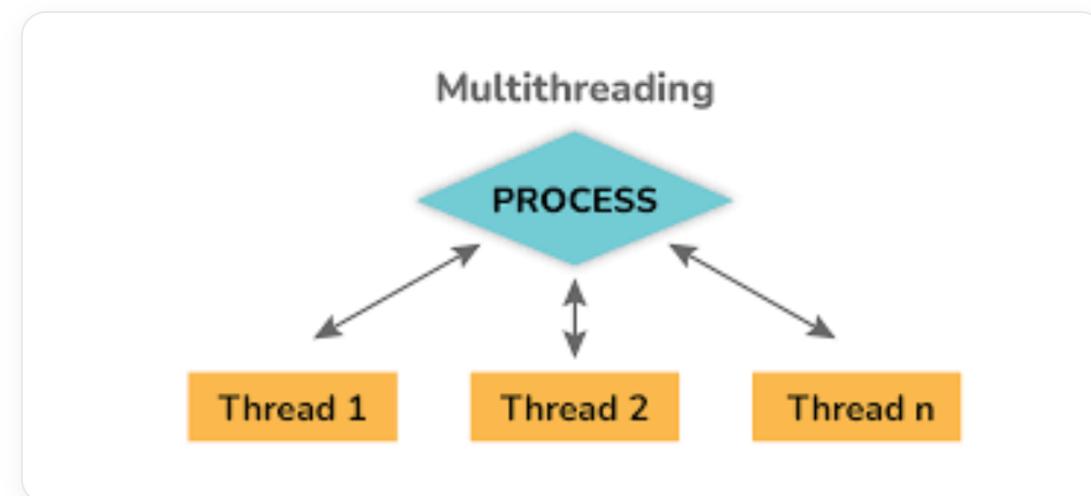
یه جور خودمونی بگم...

قبل از شروع درس امروز دوست داشتم یه دور مفاهیم رو با هم مرور کنیم که جلوتر به مشکل نخوریم. تا الان یاد گرفتیم که یک برنامه جاوا چطور نوشته میشه. به هر برنامه‌ای که در زمان اجرا داریم، میگیم **process**. نکته مهم اینه که مسئول اجرای **process**ها همون واحد پردازشی کامپیوتر، یعنی **CPU** هست.

در واقع هر برنامه به یه سری دستور به زبان‌های مثل assembly تبدیل میشه. مثلا فرض کنید برنامه ما ۵۰۰ خط دستور داره. اگه CPU تک‌هسته‌ای داشته باشیم، در هر لحظه فقط یه دستور می‌تونه اجرا بشه.

اما نکته اینجاست که کامپیوتر ما در آن واحد فقط یک برنامه رو اجرا نمی‌کنه! شما الان در حال خوندن این جزو هستید، در عین حال می‌تونید به موزیک گوش بدید (اجرای موزیک خودش یه **process** و نیاز به CPU داره)، مروگرتون هم باز باشه و...

ولی از دید شما همه این برنامه‌ها همزمان در حال اجرا هستن. پس باید CPU هی بین **process**‌های مختلف سوییچ کنه: یه مدت کوتاه روی یکی اجرا می‌کنه، بعد میره سراغ یکی دیگه. در واقع CPU بین همه **process**‌ها مشترکه، اما **در هر لحظه فقط یک دستور از یک process رو اجرا می‌کنه**.



واژه‌نامه خودمونی

- فرآیند یا برنامه(**Process**) یه برنامه در حال اجراست. مثلا وقتی موزیک پلیر رو باز می‌کنی، اون تبدیل به یه پراسس میشه که CPU باید دستوراتشو اجرا کنه.
- نخ (**Thread**) یه مسیر سبکتر و کوچیکتر از پردازش. انگار توی همون برنامه چندتا کارگر جدا داشته باشی که هر کدام یه بخش از کار رو انجام بدن.
- زیربرنامه یا زیرفرآیند (**Subprocess**) یعنی یه پراسس جدید که توسط یه پراسس دیگه ساخته میشه. مثلا وقتی برنامه‌ت یه برنامه دیگه رو صدابزن.
- وقتی CPU از اجرای یه پراسس/ترد دست می‌کشه و میره سراغ یه پراسس/ترد دیگه. مثل وقتی که معلم از سوال یکی ول می‌کنه میره جواب نفر دیگه رو بدنه.
- یعنی انجام چندتا کار با هم. در عمل CPU خیلی سریع بین کارها جابه‌جا میشه، جوری که به چشم ما میاد همه‌چی با هم در حال اجراست.

مقدمه: مالتی‌تردینگ در جاوا

مالتی‌تردینگ (Multithreading) فرآیند اجرای همزمان چندین نخ (Thread) است. نخ‌ها در واقع زیرفرآیندهای سبک هستند و کوچک‌ترین واحد پردازشی در یک برنامه محسوب می‌شوند. هدف اصلی مالتی‌تردینگ، **افزایش کارایی** و **انجام چند کار به طور همزمان** با استفاده از منابع مشترک است.

چرا از مالتی‌تردینگ استفاده می‌کنیم؟

- به جای چندین فرآیند سنگین، از نخ‌ها استفاده می‌کنیم چون نخ‌ها حافظه مشترک دارند و سبک‌تر هستند.
- جابجایی بین نخ‌ها (Context Switching) سریع‌تر از جابجایی بین فرآیندها است.
- باعث **صرفه‌جویی در حافظه** و **کاهش زمان اجرا** می‌شود.
- کاربردهای گسترده در برنامه‌های گرافیکی، بازی‌ها و اپلیکیشن‌ها دارد.

مزایای مالتی‌تردینگ در جاوا

1. برنامه هنگام اجرای یک نخ، نخ‌های دیگر را بلاک نمی‌کند.
2. امکان انجام همزمان چند عملیات را فراهم می‌کند، در نتیجه زمان کمتری مصرف می‌شود.
3. اگر یک نخ دچار خطا شود، نخ‌های دیگر مستقل عمل کرده و متوقف نمی‌شوند.

مالتی‌تسکینگ (Multitasking)

مالتی‌تسکینگ یعنی اجرای چند کار به صورت همزمان. ما از مالتی‌تسکینگ استفاده می‌کنیم تا CPU را به بهترین شکل ممکن به کار بگیریم.

انواع مالتی‌تسکینگ

1. مالتی‌تسکینگ مبتنی بر پراسس (Multiprocessing) یا Process-based Multitasking

- هر پراسس آدرس جداگانه در حافظه دارد (یعنی هر برنامه فضای حافظه مخصوص به خودش را می‌گیرد).
- پراسس‌ها سنگین (Heavyweight) هستند.
- هزینه ارتباط (Communication) بین پراسس‌ها بالاست.
- جابجایی از یک پراسس به پراسس دیگه (Context Switch) زمان بره چون باید رجیسترها، مپ‌های حافظه و لیست‌ها ذخیره و بارگذاری بشون.

2. مالتی‌تسکینگ مبتنی بر ترد (Multithreading) یا Thread-based Multitasking

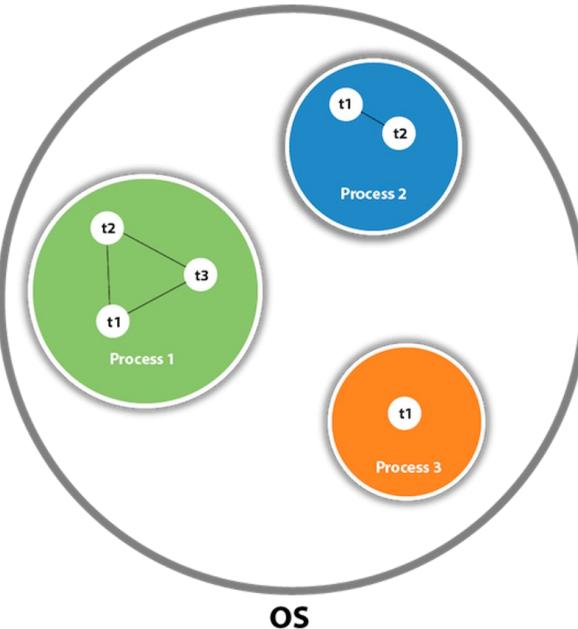
- همه تردها یک فضای آدرس مشترک دارند.
- تردها سبک (Lightweight) هستند.
- هزینه ارتباط بین تردها پایین و سریع‌تره.

ترد (Thread) در جاوا

ترد در واقع یک زیرفرآیند سبک (**Lightweight Subprocess**) و کوچک‌ترین واحد پردازشی در سیستم است. هر ترد یک مسیر جداگانه برای اجرا دارد. نخ‌ها مستقل از هم عمل می‌کنند؛ اگر در یکی خطای رخ بدهد، روی بقیه تأثیر نمی‌گذارد. همه‌ی تردها از یک ناحیه حافظه مشترک استفاده می‌کنند.

ویژگی‌های اصلی ترد

- یک ترد همیشه داخل یک پراسس اجرا می‌شود.
- بین تردها **context switching** انجام می‌شود (جابجایی سریع CPU بین نخ‌ها).
- هر سیستم عامل می‌تواند چندین پراسس داشته باشد و هر پراسس هم می‌تواند شامل چندین ترد باشد.



در این تصویر می‌بینیم که سیستم‌عامل (OS) شامل چند پراسس مختلف است (Process 1, Process 2 و Process 3). هر پراسس می‌تواند شامل یک یا چند ترد باشد. برای مثال:

- Process 1 شامل سه ترد (t1, t2, t3) است.
- Process 2 شامل دو ترد (t1, t2) است.
- Process 3 فقط یک ترد دارد.

اگر CPU ما **فقط یک هسته (Single Core)** داشته باشد، باز هم امکان اجرای چند ترد وجود دارد. در این حالت CPU به کمک context switching بین تردها جابجا می‌شود. مثلاً CPU برای چند میلی‌ثانیه دستورهای t1 را اجرا می‌کند، بعد سراغ t2 می‌رود، بعد t3 و دوباره برگرداند به t1. سرعت این جابجایی آنقدر بالاست که به چشم ما همهی تردها همزمان در حال اجرا به نظر می‌آیند.

کلاس Thread در جاوا

جاوا برای کار با نخ‌ها (**Threads**) یک کلاس به اسم **Thread** در اختیار ما قرار داده. این کلاس متدها و سازنده‌های مختلفی داره که به کمکشون می‌توانیم نخ‌ها رو ایجاد و مدیریت کنیم. در واقع **Thread** از کلاس **Object** ارث بری می‌کنه و اینترفیس **Runnable** رو هم پیاده‌سازی کرده.

کاربردهای اصلی کلاس Thread

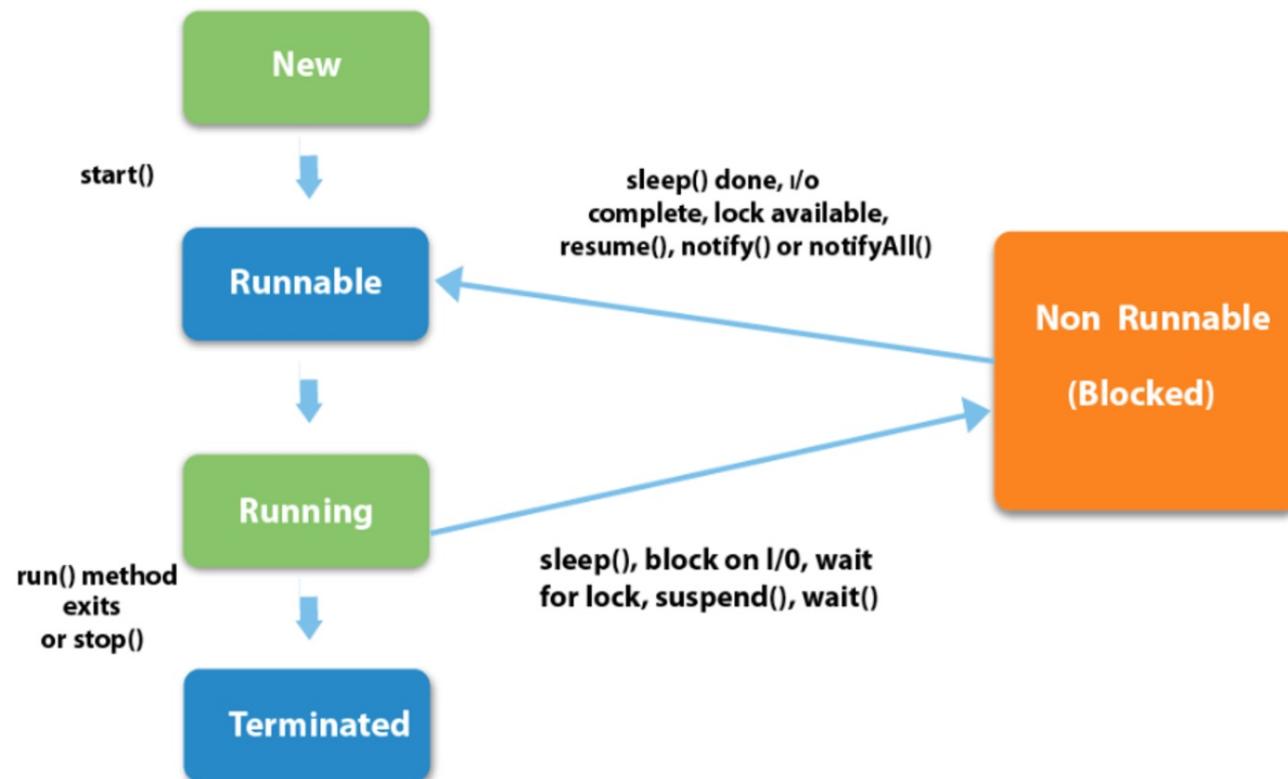
- ایجاد یک نخ جدید.
- شروع اجرای نخ با استفاده از متده **start**.
- اجرای کد داخل نخ با متده **run**.
- مدیریت ویژگی‌هایی مثل نام نخ، اولویت نخ و وضعیت اجرا.

سازنده‌های پرکاربرد Thread

- ایجاد یک نخ خالی. **Thread**
- ایجاد نخ جدید با استفاده از یک شیء که **Runnable** پیاده‌سازی کرده. **Thread(Runnable target)**
- ایجاد نخ با نام مشخص. **Thread(String name)**
- ایجاد نخ با **runnable** و نام مشخص. **Thread(Runnable target, String name)**

چرخه زندگی ترد (Thread Life Cycle)

یک نخ (Thread) در طول عمر خودش می‌توانه در وضعیت‌های مختلفی قرار بگیره. به طور کلی، چرخه زندگی نخ توسط **JVM** کنترل می‌شود. طبق تعریف Sun، چهار حالت اصلی وجود داره، اما برای درک بهتر معمولاً اون رو در پنج حالت توضیح می‌دان.



حالت‌های چرخه‌ی نخ

وقتی که یک شیء از کلاس **Thread** ساخته میشے اما هنوز متده **start** صدا زده نشده.

Thread Scheduler : بعد از اجرای متده **start** نخ آماده‌ی اجراست اما هنوز CPU بھش تخصیص داده نشده، درواقع هنوز توسط Thread Scheduler انتخاب نشده.

وقتی که Thread Scheduler نخ را انتخاب کنه و در حال اجرا باشه. یعنی CPU بھش اختصاص داده شده.

Non-Runnable (Blocked/Waiting) : نخ هنوز زنده‌ست ولی در حال حاضر نمی‌توانه اجرا بشه و CPU از دست نخ خارج شده و CPU آماده برای سرویس دادن به نخ دیگه‌ای هستش. (مثلاً با متده **sleep** یا **join** منتظر مونده).

Terminated (Dead) : وقتی که اجرای متده **run** تمام شده، نخ به وضعیت خاتمه یافته میره.

توضیح ساده با مثال

فرض کنید یک مرورگر وب (Browser) رو باز می‌کنید. وقتی تب جدید رو باز می‌کنید اما هنوز چیزی لود نشده، مثل حالتیه که نخ تازه ساخته شده (New). به محض اینکه روی یک لینک می‌کلیک و مرورگر آماده‌ی لود کردن صفحه میشے، وارد حالت Runnable شروع می‌کنه به اجرای درخواست و پردازش داده‌ها، نخ در حالت Running قرار داره. حالا اگه مرورگر منتظر پاسخ سرور بمونه یا برای لحظاتی به حالت تعليق بره (مثلاً در حال دانلود باشه)، در وضعیت Non-Runnable قرار می‌گیره. و در نهایت وقتی بارگذاری صفحه کامل شد یا تب رو بستید، اون نخ کارش تمام شده و وارد حالت Terminated میشے.

روش‌های ایجاد ترد در جاوا

در جاوا دو راه اصلی برای ایجاد و اجرای نخ وجود دارد:

۱. ایجاد نخ با ارث‌بری از کلاس Thread

در این روش یک کلاس جدید می‌سازیم که از `Thread` ارث‌بری می‌کنه و متده `run` رو بازنویسی می‌کنیم. برای شروع نخ هم از متده `start` استفاده می‌کنیم.

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Thread is running by extending Thread class...");  
    }  
  
    public static void main(String[] args) {  
        MyThread t1 = new MyThread();  
        t1.start();  
    }  
}
```

Thread is running by extending Thread class...

۲. ایجاد نخ با پیادهسازی Runnable Interface

در این روش یک کلاس می‌سازیم که **Runnable** رو پیادهسازی می‌کنه و متده **run** رو تعریف می‌کنیم. بعد باید یک شیء از **Thread** بسازیم و شیء **Runnable** رو بهش پاس بدیم.

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running by implementing Runnable interface...");
    }

    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread t1 = new Thread(myRunnable);
        t1.start();
    }
}
```

Thread is running by implementing Runnable interface...

شروع یک نخ

برای اجرای نخ باید از متد `start` کلاس `Thread` استفاده کنیم. وقتی `start` رو صدا میزنیم:

- یک نخ جدید با `call stack` جدا ساخته میشه.
- نخ از وضعیت `New` به وضعیت `Runnable` میره.
- وقتی `Scheduler` فرستت بد، متد `run` اجرا میشه.

```
class MyRunnable implements Runnable {  
    public void run() {  
        System.out.println("Thread is running...");  
    }  
  
    public static void main(String[] args) {  
        MyRunnable m1 = new MyRunnable();  
        Thread t1 = new Thread(m1);  
        t1.start();  
    }  
}
```

Thread is running...

نکته مهم

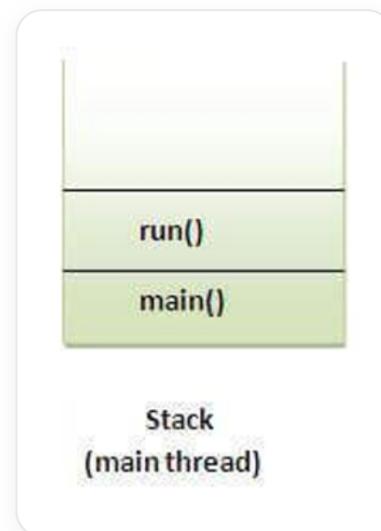
- اگه فقط کلاس رو بسازیم که Runnable رو پیاده‌سازی کرده باشه، اون شیء به خودی خود نخ حساب نمیشه.
- برای همین باید حتماً یک شیء از کلاس Thread بسازیم و شیء Runnable رو بهش پاس بدیم.
- با این کار run داخل نخ جدید اجرا میشه.

توضیح دقیق تفاوت start و run

متد **run** در واقع فقط یک متد معمولی جاواست، مثل هر متد دیگه‌ای که در کلاس‌هاتون می‌نویسید. وقتی شما مستقیم **run** را صدا می‌زنید، کدهای داخلش در همون نخ فعلی (مثلاً **main thread**) اجرا می‌شون. بنابراین هیچ نخ جدیدی ساخته نمی‌شه، هیچ call stack تازه‌ای ایجاد نمی‌شه و مدیریت حافظه هم فقط در همون نخ اصلی انجام می‌شه.

اما وقتی متد **start** را صدا می‌زنید، اتفاقات زیر رخ میده:

- JVM یک **Thread object** واقعی ایجاد می‌کنه.
- برای اون نخ جدید یک **call stack** جدا در حافظه رزرو می‌شه (یعنی فضای مخصوص خودش برای اجرای متدها داره).
- Thread Scheduler تصمیم می‌گیره چه زمانی این نخ اجرا بشه. وقتی نوبتش برسه، متد **run** درون اون نخ جدید اجرا می‌شه.
- به خاطر وجود **call stack** جدا، چند نخ می‌توان به صورت همزمان (concurrent) اجرا بشن.



چرا این تفاوت مهمه؟

فرض کنید شما می‌خوايد دو کار رو همزمان انجام بدید، مثلا:

- نخ اول: دانلود فایل از اینترنت.

- نخ دوم: پردازش داده و نمایش روی صفحه.

اگه از `run` استفاده کنید، هر دو کار پشت سر هم و روی همون `main thread` اجرا می‌شون. یعنی اولی باید تموم بشه تا دومی شروع بشه. در عمل **هیچ مالتی‌تредینگی رخ نداده!**

اما اگه از `start` استفاده کنید، برای هر کدام یک نخ واقعی ساخته می‌شه با `call stack` جدا. پس JVM می‌تونه بین اون‌ها جابه‌جا بشه و هر دو کار به صورت همزمان (یا دست‌کم به صورت شبیه‌سازی همزمان) پیش برن. نتیجه‌اش اینه که برنامه سریع‌تر، روان‌تر و بهینه‌تر کار می‌کنه.

```
class DemoThread extends Thread {  
    public void run() {  
        System.out.println("Thread is running by: " + Thread.currentThread().getName());  
    }  
  
    public static void main(String[] args) {  
        DemoThread t1 = new DemoThread();  
  
        t1.run();  
  
        t1.start();  
    }  
}
```

Thread is running by: JJava-executor-0
Thread is running by: Thread-4

متد sleep در جاوا

متد **sleep** نخ فعلى رو برای مدت مشخصی متوقف (Suspend) میکنه. این یعنی نخ همچنان زندهست، ولی برای مدت زمانی اجرا نمیشه و بعد از اون دوباره وارد صف **Runnable** میشه.

تعریف متد

```
public static void sleep(long milliseconds) throws InterruptedException •  
public static void sleep(long milliseconds, int nanos) throws InterruptedException •
```

نکات مهم

- یک متد **static** هست و همیشه روی نخ فعلى اعمال میشه.
- در طول خواب نخ، CPU میتونه به نخهای دیگه فرصت اجرا بده.
- این متد میتونه یک **InterruptedException** پرتاب کنه، پس باید یا با **try/catch** هندل بشه یا در signature متد اعلام بشه.

متد currentThread در جاوا

تعريف

متد `currentThread` یک متد `static` از کلاس `Thread` هست. این متد یک شیء از نوع `Thread` برمی‌گرداند که نمایندهٔ نخ در حال اجراست. به کمکش می‌توانیم اطلاعاتی مثل نام نخ یا ID نخ جاری را به دست بیاریم.

سینتکس

```
public static Thread currentThread •
```

کاربرد

- برای شناسایی نخ فعلی در حال اجرا.
- برای نمایش اطلاعاتی مثل `getName` یا `getId` نخ جاری.
- برای دیباگ کردن و اطمینان از اینکه کد روی کدوم نخ اجرا میشه.

مشکل صدا زدن مستقیم متدهای run

چی میشه اگه به جای start، متدهای run رو مستقیم صدا بزنیم؟

وقتی main thread رو مستقیم صدا میزنیم، نخ جدید ساخته نمیشه و همون مثل یک متدهای عادی جاوا اجرا میشه. در این حالت همهی کدها روی run میشن و خبری از context switching نیست. یعنی نخها پشت سر هم اجرا میشن نه به صورت همزمان.

```
class TestCallRun2 extends Thread {  
    public void run() {  
        for (int i = 1; i < 6; i++) {  
            try {  
                Thread.sleep(500);  
            } catch (InterruptedException e) {  
                System.out.println(e);  
            }  
            System.out.println(i);  
        }  
    }  
  
    public static void main(String[] args) {  
        TestCallRun2 t1 = new TestCallRun2();  
        TestCallRun2 t2 = new TestCallRun2();  
  
        t1.run();  
        t2.run();  
    }  
}
```

خروجی

- 1 •
- 2 •
- 3 •
- 4 •
- 5 •
- 1 •
- 2 •
- 3 •
- 4 •
- 5 •

چرا اینطوری شد؟

چون نخها به صورت واقعی اجرا نشدن و فقط مثل یک متدهای عادی روی `t1` کامل اجرا شدن و بعد کدهای `t2` پس هیچ اشتراکی در اجرای همزمان وجود نداره و **context switching** رخ نمیده.

در جاوا Thread Scheduler

وقتی چند نخ (Thread) در وضعیت **Runnable** هستن، این که دقیقا کدوم نخ اجرا بشه دست ما نیست! تصمیم‌گیری در مورد اینکه چه زمانی و کدام نخ باید اجرا بشه به عهده‌ی **Thread Scheduler** هست که بخشی از JVM محسوب میشے.

ویژگی‌های Thread Scheduler

- فقط یک نخ می‌تونه در یک لحظه روی یک پردازنده (CPU core) اجرا بشه.
- مشخص می‌کنه که در هر لحظه کدام نخ اجرا بشه.
- هیچ تضمینی وجود نداره که نخها به ترتیب شروع شدن اجرا بشن.
- الگوریتم دقیق Scheduler به پیاده‌سازی JVM و سیستم‌عامل بستگی داره.

نکته مهم

حتی اگر شما چند نخ بسازید و به ترتیب `start` صدا بزنید، خروجی همیشه همون ترتیبی که انتظار دارید نخواهد بود. این دقیقا به خاطر وجود Scheduler هست. پیشنهاد می‌شود کد زیر را در محیط IntelliJ Idea اجرا کنید.

```

class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(Thread.currentThread().getName() + " is running... step " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();

        t1.start();
        t2.start();
        t3.start();
    }
}

```

خروجی احتمالی

- ...Thread-0 is running •
- ...Thread-1 is running •
- ...Thread-2 is running •

یا حتی ممکنه ترتیب فرق کنه: مثلا اول Thread-0 اجرا بشه بعد Thread-2. چون انتخاب نخها در هر لحظه دست Scheduler هست.

آیا می‌توان یک نخ را دوبار استارت کرد؟

پاسخ

خیر وقتی یک نخ با `start` آغاز می‌شود، دیگه نمی‌توانیم دوباره اون رو استارت کنیم. اگر این کار رو بکنیم، یک خطای `IllegalThreadStateException` رخ میدهد.

چرا؟

چون وقتی نخ برای اول استارت می‌شود، JVM یک `call stack` جدید برای اون نخ ایجاد می‌کند. بعد از اینکه نخ به وضعیت `Terminated` برسد، دیگه نمی‌توانیم همون نخ رو زنده کنیم. در واقع هر نخ فقط یک بار می‌توانه از `New` به `Runnable` بره.

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Thread is running...");  
    }  
  
    public static void main(String[] args) {  
        MyThread t1 = new MyThread();  
        t1.start();  
        t1.start();  
    }  
}
```

```
Thread is running...  
-----  
java.lang.IllegalThreadStateException: null  
    at java.base/java.lang.Thread.start(Thread.java:1534)  
    at MyThread.main(#17:9)  
    at .(#28:1)
```

نکته مهم

اگر نیاز دارید دوباره کاری مشابه انجام بشه، باید یک شیء نخ جدید بسازید. یعنی به جای اجرای دوباره `t1.start` باید یک شیء جدید مثل `new MyThread` بسازید و اون رو اجرا کنید.

متد join در جاوا

تعريف

متد **join** باعث میشه نخ فعلى منتظر بمونه تا نخ دیگری که روی اون صدا زده شده کارش رو تموم کنه. به بیان ساده: اگه نخ A روی نخ B متد **join** رو صدا بزنه، نخ A متوقف میشه تا B کامل اجرا بشه.

سینتکس

```
public final void join throws InterruptedException •  
public final void join(long millis) throws InterruptedException •  
public final void join(long millis, int nanos) throws InterruptedException •
```

```
class JoinExample extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(Thread.currentThread().getName() + " - Step " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }

    public static void main(String[] args) {
        JoinExample t1 = new JoinExample();
        JoinExample t2 = new JoinExample();
        JoinExample t3 = new JoinExample();

        t1.start();
        try {
            t1.join();
        } catch (InterruptedException e) {
            System.out.println(e);
        }

        t2.start();
        t3.start();
    }
}
```

خروجی احتمالی

ابتدا همهی مراحل نخ t1 اجرا میشه (چون main منتظرش). بعد از اتمام t1، نخهای t2 و t3 همزمان شروع میشن:

- Thread-0 - Step 1 •
- Thread-0 - Step 2 •
- Thread-0 - Step 3 •
- Thread-1 - Step 1 •
- Thread-2 - Step 1 •
- ... •

نام‌گذاری نخ‌ها در جاوا

نام پیش‌فرض نخ‌ها

هر نخ در جاوا به صورت پیش‌فرض یک نام داره که توسط JVM تعیین می‌شه. معمولاً به صورت Thread-0، Thread-1 و ... نام‌گذاری می‌شه.

تغییر نام نخ

ما می‌توانیم با استفاده از متدهای getName و setName نام فعلی نخ را بخونیم.

سینتکس

```
public String getName  
• برگرداندن نام نخ
```

```
public void setName(String name)  
• تغییر نام نخ
```

```
public long getId  
• برگرداندن شناسه‌ی نخ
```

```

class TestThreadNaming extends Thread {
    public void run() {
        System.out.println("running...");
    }

    public static void main(String[] args) {
        TestThreadNaming t1 = new TestThreadNaming();
        TestThreadNaming t2 = new TestThreadNaming();

        System.out.println("Name of t1: " + t1.getName());
        System.out.println("Name of t2: " + t2.getName());
        System.out.println("ID of t1: " + t1.getId());

        t1.start();
        t2.start();

        t1.setName("CustomThread");
        System.out.println("After changing name of t1: " + t1.getName());
    }
}

```

خروجی احتمالی

Name of t1: Thread-0 •

Name of t2: Thread-1 •

ID of t1: 8 •

...running •

After changing name of t1: CustomThread •

...running •

نکته‌ها

- یک عدد long مثبت هست که هنگام ساخت نخ ایجاد میشه.

- در طول عمر نخ ثابت و یکتا باقی می‌مانه.

- بعد از شدن نخ، ممکنه JVM اون ID را برای نخ‌های جدید دوباره استفاده کنه.

اولویت‌بندی نخ‌ها (Thread Priority)

تعريف

هر نخ در جاوا یک **اولویت (Priority)** داره که عددی بین 1 تا 10 هست. به صورت پیش‌فرض همه نخ‌ها اولویت 5 دارن (این عدد در واقع فقط یک **پیشنهاد** به Thread Scheduler به `Thread.NORM_PRIORITY` است). این عدد را در نظر بگیره.

نکته مهم

اولویت نخ به تنهایی تعیین‌کننده نیست. حتی اگه شما یک نخ رو با `MAX_PRIORITY` بسازید، هیچ تضمینی وجود نداره که همیشه زودتر از نخ‌های دیگه اجرا بشه. چون اجرای واقعی نخ‌ها به **Thread Scheduler** وابسته‌ست و هم توسط JVM و سیستم‌عامل مدیریت میشه.

بنابراین:

- نخ با Priority بالاتر ممکنه دیرتر از نخ با Priority پایین‌تر اجرا بشه.
- رفتار دقیق نخ‌ها روی سیستم‌عامل‌های مختلف (مثل ویندوز، لینوکس یا مک) ممکنه متفاوت باشه.
- از Priority بیشتر برای «پیشنهاد دادن» استفاده میشه، نه برای کنترل قطعی ترتیب اجرای نخ‌ها.

مقدارهای ثابت پیشفرض

```
class PriorityExample extends Thread {  
    public void run() {  
        System.out.println("Running thread: "  
            + Thread.currentThread().getName()  
            + " with priority: "  
            + Thread.currentThread().getPriority());  
    }  
  
    public static void main(String[] args) {  
        PriorityExample t1 = new PriorityExample();  
        PriorityExample t2 = new PriorityExample();  
        PriorityExample t3 = new PriorityExample();  
  
        t1.setName("LowPriorityThread");  
        t2.setName("NormalPriorityThread");  
        t3.setName("HighPriorityThread");  
  
        t1.setPriority(Thread.MIN_PRIORITY); // 1  
        t2.setPriority(Thread.NORM_PRIORITY); // 5  
        t3.setPriority(Thread.MAX_PRIORITY); // 10  
  
        t1.start();  
        t2.start();  
        t3.start();  
    }  
}
```

(پیشفرض همه نخها)

Thread.MIN_PRIORITY = 1 •
Thread.NORM_PRIORITY = 5 •
Thread.MAX_PRIORITY = 10 •

متدهای مربوط

دریافت اولویت نخ → public final int getPriority •

تغییر اولویت نخ → public final void setPriority(int priority) •

نکته مهم

اولویت نخ فقط یک پیشنهاد به Thread Scheduler هست. هیچ تضمینی وجود نداره که همیشه نخ با Priority بالاتر زودتر اجرا بشه. در واقع نتیجه نهایی به **JVM** و **سیستم‌عامل** بستگی داره.

مطالعه بیشتر: الگوریتم‌های زمان‌بندی در Scheduler

First Come First Serve (FCFS)

ساده‌ترین الگوریتم زمان‌بندی. نخ‌ها به ترتیب که درخواست اجرا شدن وارد صف می‌شون، همواره ترتیب هم توسط CPU اجرا می‌شون. مثل صف نانوایی: هر کی زودتر بیاد، زودتر نوبتش می‌شود.

Shortest Job Next (SJN) / Shortest Job First (SJF)

در این الگوریتم، نخ‌هایی که کار کمتری دارن (یعنی زمان اجرای کمتری می‌برن) زودتر انتخاب می‌شون. مشکلش اینه که نخ‌های طولانی ممکنه مدت زیادی پشت صف بموون (Starvation) شوند.

Round Robin (RR)

هر نخ یک بازه زمانی مشخص (Quantum Time Slice) یا (Time Slice) داره. وقتی زمانش تمام شد، CPU نخ بعدی رو انتخاب می‌کنه. این چرخه همینطور ادامه پیدا می‌کنه تا همه نخ‌ها فرصت اجرا داشته باشن. این روش خیلی شبیه به **context switching** در سیستم‌های واقعی هست.

در این روش هر نخ یک اولویت (Priority) داره و Scheduler نخها رو براساس اون انتخاب می‌کنه. نخهایی با Priority بالاتر احتمال بیشتری دارن که زودتر اجرا بشن. اما یک نکته مهم: همیشه هم نخ با Priority بالاتر اجرا نمیشه، چون در عمل Scheduler باید تعادل بین نخها رو حفظ کنه.

مثال: الگوریتم لاتاری (Lottery Scheduling)

یکی از پیاده‌سازی‌های جالب Priority Scheduling الگوریتم لاتاری هست. توی این روش، هر نخ براساس Priority که داره یک تعداد بلیت لاتاری دریافت می‌کنه. Scheduler در هر بار انتخاب، به صورت تصادفی یک بلیت رو برمی‌داره و نخ صاحب اون بلیت اجرا میشه.

برای مثال:

- نخ A با Priority=3 یعنی 3 بلیت دریافت می‌کند.
- نخ B با Priority=1 یعنی 1 بلیت دریافت می‌کند.
- نخ C با Priority=6 یعنی 6 بلیت دریافت می‌کند.

در این حالت احتمال انتخاب نخ C بیشتر از بقیه است، اما هنوز هم امکان داره نخ A یا B انتخاب بشن. این روش باعث میشه هم نخهای مهم‌تر شанс بیشتری داشته باشن و هم نخهای کم‌اهمیت کامل کنار گذاشته نشن.

خودمون رو بسنجیم

این بخش برای این طراحی شده که بعد از مطالعه مباحث مالتی‌تردینگ، خودت رو محک بزنی و مطمئن بشی مفاهیم کلیدی رو یاد گرفتی. بدون نگاه کردن به متن درس، به پرسش‌ها جواب بد:

- تفاوت **thread** و **process** چیه؟ از نظر حافظه و Context Switching توضیح بد.
- تفاوت اصلی **start** و **run** در اجرای نخ‌ها چیه؟
- چرخه‌ی زندگی یک نخ در جاوا شامل چه حالت‌هایی میشه؟ برای هر حالت یک مثال ساده بزن.
- اگه یک نخ رو دوبار **start** کنیم چه اتفاقی میفته؟ چرا؟
- متد **sleep** چه کاری انجام میده و چه تاثیری روی نخ‌های دیگه داره؟
- کاربرد متد **join** چیه و چه موقع استفاده میشه؟
- چطور می‌تونیم نام یک نخ رو تغییر بدیم؟ چه متدهایی برای این کار وجود دارن؟
- خروجی متد **currentThread** چی رو نشون میده؟
- نخ‌ها چه کاربردی داره؟ آیا نخ با Priority بالاتر همیشه زودتر اجرا میشه؟ چرا؟

پایان

در صورت هرگونه سوال یا پیشنهاد می‌توనی با من
در ارتباط باشی:

gmail: arasvalizadeh@gmail.com
telegram: @arasvalizadeh