Concurrency

تا به این مرحله در رابطه با برنامه نویسی مرحله ای (sequential programming) آموخته اید.
هر چیزی در یک برنامه به صورت مرحله ای از یک زمان رخ می دهد.

معرفی

- برنامه نویسی موازی می تواند پیشرفت های بزرگی را در زمینه سرعت اجرای برنامه یا ایجاد مدلی ساده تر جهت طراحی نوع خاصی از برنامه ها و یا هر دوی این موارد،ایجاد کند.
- جاوا یک زبان multithread است و مسائل مربوط به همزمانی،چه از آن ها آگاه باشید یا نباشید،همیشه موجودند.

جنبه های مختلف همزمانی

- مسائلی که از طریق همزمانی به حل آن ها می پردازیم به دو دسته ”سرعت“ و ”مدیریت طراحی“ تقسیم می شوند.

✓ اجرای سریع تر:

- مسئله سرعت در ابتدا ساده به نظر می رسد.در صورتی که می خواهید برنامه ای سریع تر اجرا شود،آن را به چند قطعه تقسیم کنید و هر قطعه را در یک پردازشگر جدا،اجرا نمایید.
- جاوا از رویکرد سنتی پشتیبانی از امکانات thread در بالای یک زبان sequential استفاده می کند.
- به جای انشعاب پردازش های خارجی در سیستم عامل های چند وظیفه ای(multitasking operating system) ، این عملیات را در طول یک پردازش در برنامه اجرایی،ایجاد می کند.

مفاهیم پایه multithreading

- برنامه نویسی همزمان این امکان را فراهم می کند یک برنامه را به چند قطعه جدا، با وظایف اجرایی مستقل،تقسیم کنیم.با استفاده از multithreading هر کدام از این وظایف مستقل توسط یک thread اجرایی،راه اندازی می‌شود:
  - تعریف وظایف
  - یک thread یک وظیفه را راه اندازی می کند.بنابراین نیازمند روشی برای توصیف این وظایف هستیم و این کار از طریق واسط Runnable انجام می شود.

مفاهیم پایه multithreading

```
public class LiftOff implements Runnable{
    protected int countDown = 10; // Default
    private static int taskCount = 0;
    private final int id = taskCount++;
    public LiftOff() {}
   public LiftOff(int countDown){
        this.countDown = countDown;
    }
    public String status(){
        return "#" + id + "(" +
               (countDown > 0 ? countDown : "Liftoff!") + "), ";
    }
    public void run(){
        while(countDown-- > 0){
            System.out.print(status());
             Thread.yield();
        }
    }
}
```

کلاس Thread

- روش سنتی برای تبدیل یک شی Runnable به یک وظیفه ی کاری این است که آن را در constructor یک thread نگه داری.مثال زیر نشان می دهد که چگونه یک شی Liftoff از یک thread استفاده می کند.

```
public class BasicThreads{
    public static void main(String[] args){
        Thread t = new Thread(new LiftOff());
        t.start();
        System.out.println("Waiting for LiftOff");
    }
}
```

استفاده از Executorها

- اجرا کننده های java.util.concurrent در Java SE5 برنامه نویسی همزمان را از طریق مدیریت اشیای Thread ،آسان می کند.
- می توان به جای آنکه اشیای Thread را به صورت صریح ایجاد کرد،از Executor استفاده نمود.

```
import java.util.concurrent.*;
public class CachedThreadPool{
    public static void main(String[] args){
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
           exec.execute(new LiftOff());
        exec.shutdown();
    }
}
```

- CachedThreadPool در هر Task یک thread.ایجاد می کند

استفاده از Executorها

- می توان به آسانی CachedThreadPool در مثال قبل را با نوع متفاوتی از Executor جایگزین نمود.
- FixedThreadPool مجموعه محدودی ازthreadها را برای اجرای وظایف معین شده،استفاده می کند .
- SingleThreadExecutor همانند FixedThreadPool است اما با سایز یکthread و زمانی مفید است که بخواهیم چیزی را در یکthread به طور مستمر اجرا کنیم .

قرار دادن thread در موقعیت sleep

- یک راه ساده برای تاثیر گذاشتن بر روی شیوه رفتار وظایف این است که sleep() را فراخوانی کنیم و اجرای وظیفه مورد نظر را برای مدت زمانی متوقف کنیم.

```
import java.util.concurrent.*;
public class SleepingTask extends LiftOff{
    public void run(){
        try{
            while(countDown-- > 0){
                System.out.print(status());
                // Old-style:
                // Thread.sleep(100);
                // Java SE5/6-style:
                TimeUnit.MILLISECONDS.sleep(100);
            }
        }
        catch(InterruptedException e){
            System.err.println("Interrupted");
        }
    }
```

شیوه های مختلف کدنویسی

- در همه مثال هایی که تاکنون دیده اید،کلاس های task،همه runnable را پیاده سازی می کنند. در موارد ساده تر می توان از شیوه ی دیگری استفاده کرد و مانند مثال زیر به طور مسقیم از Thread ارث برد:

```
public class SimpleThread extends Thread{
    private int countDown = 5;
    private static int threadCount = 0;
    public SimpleThread() {  // Store the thread name:
         super(Integer.toString(++threadCount));
        start();
    }
    public String toString(){
        return "#" + getName() + "(" + countDown + "), ";
    }
    public void run(){
        while(true){
            System.out.print(this);
            if(--countDown == 0)
                return;
        }
    }
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++)
            new SimpleThread();
    }
}
```

حل مسائل ارتباطی مربوط به منابع به اشتراک گذاشته شده

- برای جلوگیری از وقوع تصادم در منابع،جاوا داری ساختار پشتیبانی داخلی در قالب کلیدواژه synchronized می باشد.هنگامی که می خواهیم taskای را اجرا کنیم،قطعه کدی با کلیدواژه synchronized تضمین می شود و چک می کند تا در صورتی که قفل مرتبط با آن در درسترس بود،آن را بدست آورده،اجرا می کند و سپس آن را انتشار می دهد.
- منابع مشترک معمولا قطعه ای از حافظه در قالب یک object هستند،اما می توانند یک فایل یا یک پورت ورودی/خروجی،یا چیزی مانند پرینتر هم باشند.

```
synchronized void f(){
    /* ... */
}
```

قسمت های بحرانی

- گاهی اوقات می خواهیم مانع از دسترسی چند thread به قسمتی از کد یک متد به جای کل متد،شویم.قسمتی از کد که می خواهیم بدین شیوه جداسازی کنیم،یک قسمت بحرانی(Critical sections) نامیده می شود و از طریق استفاده از کلید واژه synchronized ایجاد می شود.در این حالت کلید واژه synchronized جهت تخصیص objectای که قفل آن مورد استفاده قرار گرفته است ، استفاده می شود و قطعه کد جداشده را synchronize می کند.

```
synchronized(syncObject){
// This code can be accessed
// by only one task at a time
}
```

Thread local storage

- حافظه محلی thread (Thread local storage یا TIS) مکانیزمی است که به صورت خودکار حافظه های مختلفی را برای یک متغیر، به ازای هر کدام از thread های مختلف که از object مد نظر استفاده می کنند،ایجاد می کند.
- بنابراین در صورتی که پنج thread داشته باشید که از یک object با متغیر x استفاده می کنند، حافظه محلی thread پنج قطعه حافظه مختلف برای x ایجاد می کند.

```
private static ThreadLocal<Integer> value = new ThreadLocal<Integer>();
public static void increment(){
    value.set(value.get() + 1);
}
public static int get(){
    return value.get();
}
```

وضعیت های thread

- **New** یک thread: تنها برای یک لحظه در این حالت باقی می ماند و آن هنگامی است که thread ایجاد می شود و منابع سیستمی مورد نیاز را تخصیص داده مقدار دهی اولیه را انجام می دهد و در این مرحله واجد شرایط دریافت زمان CPU. می شود

Runnable این به این معنی است یک thread: زمانی می تواند اجرا شود که بازه زمانی time-slicing دارای (چرخه های CPU برای thread. مدنظر باشد

- **Blocked** thread: می تواند اجرا شود،اما چیزی مانع از اجرای آن می شود. در زمانی که thread در حالت blocked(قرار دارد، زمانبند scheduler به سادگی آن را (skip می کند و هیچ زمان cpu ای را به آن اختصاص نمی دهد .
- **Dead** thread: ای که در وضعیت Dead یا terminated قرار دارد دیگر قابل زمانبندی نیست و هیچ زمان cpu ای برای آن در نظر گرفته نمی شود. وظیفه این thread. پایان یافته و دیگر قابل اجرا نیست

وقفه

- می توان وظیفه ای که در وضعیت blocked قرار دارد را پایان داد.
- کلاس Thread شامل متدی به نام interrupt() می باشد.این متد وضعیت وقفه را به thread اعمال می کند.thread ای با وضعیت وقفه، در صورتی که در وضعیت blocked قرار داشته باشد و یا در حال فعالیتی باشد که باعث block شود، استثنای InterruptedException را throw خواهد کرد.

همکاری بین taskها

- **wait() notifyAll**

wait این امکان را می دهد که برای تغییر در شرایطی که خارج از کنترل ( ) نیروهای موجود در متد فعلی است،صبر کنیم.معمولا این شرایط توسط یک task دیگر ،تغییر خواهد کرد.

- ✓ زمانی که یک notify( ) یا notifyAll( ) رخ می دهد،فرض می شود اتفاق خوشایندی رخ داده است.بنابراین task بیدار شده و تغییرات را چک می کند.

همکاری بین taskها

- توجه به این نکته مهم است که هنگامی که متد sleep() فراخوانی می شود،قفل object را آزاد نمی کند.به بیان دیگر،هنگامی که taskای در داخل متد وارد قسمت wait() می شود،اجرای thread به حالت تعلیق درآمده وقفل آن object آزاد می شود.
- دو نوع wait() وجود دارد.در نوع اول متد یک آرگومان بر حسب میلی ثانیه را به عنوان ورودی می گیرد و مفهوم آن همانند متد sleep() است: "برای این مدت زمان متوقف بشو"
- یکی از جنبه های نسبتا منحصر به فرد notify،()wait() و notifyAll() این است که همه این متدها بخشی از کلاس پایه Object هستند نه بخشی از Thread.

بن بست (Deadlock)

- دریافتیم که یک object می تواند متدهایی به صورت synchronized یا شیوه های دیگر قفل شدن را داشته باشد که مانع از دسترسی taskها به آن object می شوند.همچنین آموختیم که taskها می توانند در وضعیت blocked قرار گیرند.
- بنابراین ممکن است که task ای مجبور به صبر کردن برای task دیگری باشد و آن task دوم هم منتظر taskای دیگر باشد و همین طور تا آخر.بنابراین با یک حلقه مداوم از taskهایی روبرو خواهیم شد که هر task منتظر دیگری است و هیچ کدام نمی توانند حرکت کنند.این حالت بن بست (deadlock) نامیده می شود.