Multi-Threading

وأخيراً وصلنا إلى آخر بحث وأصعب بحث، وهو يكتسب أهمية خاصة من كون هذه السنة هي أول سنة يعطى فيها للطلاب، لذا أرجو أن أوفق لإفادتكم قدر المستطاع، ولكن المشكلة أن هذا البحث ليس كغيره إذ أنه يصعب كثيراً -إن لم نقل يستحيل- فيه بشكل نظري، ولكن يجب أن نبرمج أكثر من برنامج بالاعتماد على هذه التقنية حتى نترسخ مفاهيمها في العقل..

يمكنكم اعتبار هذا البحث وكأنه تتعلم طريقة جديدة للتفكير البرمجي، إذ أننا تعودنا حتى الآن على التفكير بالبرنامج بشكل تسلسلي ومتتالي، ولم يخطر في بالنا أن نرى جزأين من الكود يعملان معاً!

- إذاً ما هو الـ thread ؟

هو في الحقيقة برنامج فرعي يعمل على التوازي مع برنامج رئيسي بحيث تنفذ تعليماتهما في نفس الوقت! ويمكن زيادة عدد هذه البرامج الفرعية بالقدر الذي نشاء بحيث يصبح برنامجنا وكأنه مجموعة من البرامج التي تعمل معاً في نفس الوقت.

- ولكن كيف يحدث هذا ؟

في الحقيقة الأمر مختلف بعض الشيء عما تعلمناه في مادة بنيان الحواسيب عندما درسنا أوامر المقاطعة، إذ أن للمعالج رتلاً من الـ threads التي تنتظر دورها لتنفذ، ويقوم المعالج بتوزيع جهده عليها كلها بحيث يبدو في الواقع وكأن المعالج ينفذ جميعاً مع بعضها.

- كيف يمكن السيطرة على الـ thread ؟

في الحقيقة يستحيل السيطرة على الـ threads، وذلك لأنها تصبح منذ إنشائها برامج مستقلة، طبعاً هذا لا يعني أنني لن أستطيع حتى بدأ، ولكنني لم أعد قادراً على توقع ما قد ينتج عن عمل الـ thread جنباً إلى جنب مع البرنامج الرئيسي إذ أنه لا توجد أي طريقة لأعرف أي منهما يعمل الآن..

فمثلاً إذا كان لدي 2 threads أيهما نفذ على نفس المتحول لن أستطيع أن أخمن قيمة المتحول، لأنني لا أعلم أياً منهما سينفذ أولاً حتى أن تنفيذ البرنامج نفسه قد يختلف من مرة لأخرى..

والأنكى من هذا وذاك هو استحالة عمل (debugging) لأي برنامج يحوي threads لأنها تنفذ مع بعضها، ولا قدرة للـ debug إلا على متابعة أحدها فقط..

- ما الحل إذاً ؟

بعد كل هذا الكلام المقلق لا داعي للخوف، إذ أنه لا يجدر بنا أصلاً استخدام هذه التقنية في الأماكن التي لا نستطيع فيها السيطرة عليه. كما أن هناك ما يكفي من التقنيات التي تخولنا التحكم بالـ threads.

آلية عمل الـ thread:

سنتعلم كيف ننشئ thread وما هي التوابع التي يمكننا أن نتعامل معه عن طريقها، ومن ثم نعود لضرب أمثلة واقعية عليه.

- إنشاء الـ thread:

إن الـ thread في الحقيقة هو class ولكنه يملك إمكانية التشغيل الذاتي، ولكن كيف نعطي الـ class هذه القدرة ؟؟

حتى ننشئ thread يجب أن نرث من الصف Thread وأن نعيد تعريف التابع ()run.

هذا الصف هو الـ thread الحقيقي، وحتى نستطيع جعل class ما thread يجب أن يرث من هذا الصف، ويعيد تعريف التابع ()run.

إن التعليمات التي سنكتبها ضمن التابع ()run هي التي ستنفذ عند تشغيل الـ thread.

- ولكن كيف نشغل الـ thread؟

يتم تشغيل الـ thread عن طريق التابع ()start الموجود ضمن الصف Thread والذي يقوم تلقائياً بإنشاء thread وتشغيل تابع ()run له.

- وماذا عن إنهاء عمل الـ thread؟

ينتهي الـ thread عند انتهاء تنفيذ جميع التعليمات ضمن التابع ()run، وحتى نفهم الفكرة أكثر يجب أن نعلم أن البرنامج الرئيسي هو thread وأن التابع ()main مقابل تماماً للتابع ()run، فكما أن البرنامج الرئيسي ينتهي مع نهاية التابع ()main، كذلك الـ thread ينتهي مع نهاية التابع ()run.

مثال:

```
public class SimpleThread extends Thread {
    private int countDown = 50;
    private static int threadCount = 0;
    public SimpleThread() {
        super(String.valueOf(++threadCount));
        start();
    }

    public String toString() {
        return "#" + getName() + ": " + countDown;
    }

    public void run() {
        while (true) {
            System.out.println(this);
            if (--countDown == 0)return;
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 5; i++)
            new SimpleThread();
    }
}
```

لندرس المثال:

جعلنا الصف SimpleThread يرث من الصف Thread ويعيد تعريف التابع run()، ويتم تشغيل الـ thread من باني الصف SimpleThread، كما نلاحظ أن هذا الباني يستدعي باني الصف Thread ويمرر له اسمه.

ضمن التابع run() نلاحظ وجود حلقة غير منتهية تكسر عن طريق شرط if الذي بداخلها، وفي الحقيقة هذه هي الطريقة الوحيدة التي تجعل التابع run() يستمر في العمل، لأنه لولا هذه الحلقة لانتهى التابع مباشرة وبالتالي دمر الـ thread.

أنشأنا 5 threads وتركنا كل منهم يعمل على راحته، وسيطبع كل منهم اسمه 50 مرة:

#1: 50 .. #1: 1

وهكذا من أجل كل thread، ولكن الظريف أننا لن نستطيع تخمين ترتيب طباعة هذه الجمل على الخرج، لأننا لا نعلم لمن سيسمح المعالج بالعمل قبل الآخر وفي أي لحظة سيقاطع الـ thread ويسمح لغيره بالعمل...

نستخلص من المثال السابق بنتائج:

1. إن كل مرة نشغل فيها البرنامج السابق ستعطي خرجاً مختلفاً عن المرة السابقة، إذ أن المعالج يمتلك زمن انتظار تصطف فيه الـ threads وتنتظر دورها، ومهمة المعالج هي السماح لكل thread بالعمل جزءاً من الوقت، ومن ثم مقاطعته وإرساله إلى آخر الزمن والسماح للـ thread الذي يليه وهكذا..
2. إن التابع run() يحتوي غالباً على حلقة تكرار، وما إن تنتهي حتى يكون عمر الـ thread قد انتهى، لذا نستطيع الاستفادة من هذه الميزة وإنهاء عمل الـ thread بدوياً من الخارج كما يلي:

مثال:

```
public class SimpleThread extends Thread
{
    private boolean ok = true;

    public void turnOff() {
        ok = false;
    }

    public void run() {
        while (ok) {

        }
    }
}
```


يمكن الآن لأي class يملك مؤشراً على هذا الغرض أن ينهي عمل الـ thread عن طريق استدعاء التابع turnoff وعندها سينهي الـ thread دورته الحالية، ولن يفلح في الدخول في دورة جديدة، بسبب انكسار شرط الحلقة.

يمكن القول بأن هذه أسلم طريقة لإنهاء عمل الـ thread، إذ أنها تضمن إنهاء الدورة الحالية وتنفيذ ما يلي تعليمة الـ while من تعليمات، والخروج نظامياً من التابع run().

يوجد طريقة أخرى غير مجدية وهي عن طريق التابع stop() الموجود في الصف Thread وبالتالي في الصف SimpleThread الذي يرث من Thread، وسبب خطورة هذه الطريقة أنها تنهي الـ thread مباشرة حتى بدون أن تسمح له بإتمام المهمة التي بدأ بها، وهذا خطير لأن الـ thread قد يتعامل مع ملفات مثلاً ضمن حلقة لـ while ومن ثم يقوم بإغلاقها بعد انتهاء الحلقة، لكن استدعاء التابع stop() لن يسمح بهذا.

إمكانيات أخرى:

لدينا مجموعة من التوابع التي يقدمها الصف Thread والتي تتحكم بالـ thread وسنستعرض بعضها:

- sleep(): يقوم هذا التابع (بإدارة) الـ thread مقدار من الزمن يقدر بالميلي ثانية (ms)، ولكنه في الحقيقة ينوم الـ thread لهذا الزمن (على الأقل) أي الـ thread عندما يستيقظ سيجد نفسه في نهاية زمن المعالج، وبالتالي يحتاج إلى وقت حتى يعود للعمل به.

ولكن ما الفائدة من هذا التابع؟

ربما أحب المبرمج أن يفسح المجال لـ threads أكثر أهمية، لذا يريح المعالج من هذا الـ thread لفترة زمنية معينة.

إن التابع السابق يجبرك على معالجة اعتراض من النوع InterruptedException وهذا ما سنتعرف سببه بعد قليل.

- interrupt(): يقوم بقطع غفوة الـ thread وإعادته إلى العمل، وعندها سيقوم التابع sleep() برمي اعتراض من النوع InterruptedException يسبب غفوته ليعود الـ thread إلى آخر الزمن، وهذا يفسر إجبار التابع sleep() للمبرمج على معالجة هذا النوع من الاعتراضات.
- yield(): هذا التابع يقوم بنقل الـ thread إلى آخر زمن الانتظار دون أن ينومه.

يستخدم هذا التابع عندما يشعر المبرمج بأن الـ thread اكتفى بتنفيذ هذا القدر من التعليمات وعليه أن يفسح المجال لغيره.

- suspend(): يوقف عمل الـ thread يمون تحديد مدة معينة، ويتم إعادة تشغيله عن طريق التابع resume().

طريقة جديدة:

تعلمنا كيف ننشئ thread عن طريق جعل الصف يرث من Thread، وبما أن Java لا تسمح بالوراثة المتعددة فإننا حكمنا على هذا الصف بأن يكون thread فقط وهذا مزعج لأننا قد نرغب في كون الصف ذا مهمات متعددة ويرث من صف آخر غير Thread وفي أغلب الوقت نرغب في أن يكون Thread فما الحل؟

الحل يكون عن طريق تحقيق الـ interface (Runnable) والتي تحوي التابع run() وبالتالي سنضطر لإعادة تعريفه.

لكن المشكلة هنا أننا لن نستطيع التعامل مع هذا الـ class مباشرة على أنه thread وإنما سنضطر إلى إنشاء غرض من الصف Thread وتمرير مؤشر على صفنا إلى باني هذا الغرض كما يلي:

مثال:

```
public class RunnableThread implements Runnable
{
    private int countDown = 50;
    private boolean ok = true;

    public String toString() {
        return "#" + Thread.currentThread().getName() + ": " + countDown;
    }

    public void turnOff() {
        ok = false;
    }

    public void run() {
        while (true) {
            System.out.println(this);
            if (--countDown == 0)return;
        }
    }

    public static void main(String[] args) {
        Thread t;
        for (int i = 1; i <= 5; i++) {
            t = new Thread(new RunnableThread(), "" + i
            t.start();
        }
    }
}
```


فكرة:

من الممكن أيضاً أن نعرف Inner class ونجعله يرث من الصف Thread وبالتالي يكون هو فقط الـ thread وليس كل الصف. وهذه من أبي المتواضع أفضل طريقة.

التزامن synchronized:

ليكن لدينا class يحوي متحولاً i قيمته 200 ويعطينا تابعاً لزيادة i وآخر لإنقاصها.

وليكن لدينا 2 thread واحد يزيد i 100 مرة والثاني ينقصها 100 مرة.

إن طباعة i بعد انتهاء عمل الثريدين يجب أن تعطينا 200.

ولكن قد نفاجأ أحياناً بأن قيمة i ليست 200، فما السبب يا ترى؟

السبب يكمن بأن زيادة قيمة i –حركتك إنقاصها– تحتاج إلى 3 تعليمات Assembly:

```
LD i
inc i, 1
ST i
```

لنفرض أن الـ thread المسؤول عن الزيادة نفذ عملية LD فوجد أن قيمة i = 200، ولكن المعالج قام بمقاطعة عمله قبل أن تتم الزيادة ليعطي الدور للـ thread المسؤول عن الإنقاص والذي وجد أيضاً أن قيمة i = 200 فأنقصها إلى 199.

عندما سيعود الدور إلى الـ thread الأول سيقوم بزيادة i ولكن اعتماداً على القيمة التي أصبحت عنده نتيجة عملية LD علماً أنها ليست آخر قيمة لـ i وعندها سيحصل فرق في القيمة النهائية لـ i.

فما الحل؟

الحل يكون بأن يمنع كل thread أي thread آخر من الاقتراب من الغرض الذي يحوي i طوال فترة تنفيذه لتابع الزيادة أو النقصان، وبالتالي نضمن عدم حدوث مشاكل في i.

هذه العملية تشبه وضع قفل على الغرض من قبل الـ thread إلى أن ينتهي من جميع عملياته عليه ومن ثم تحريره ليتمكن الآخرون من استخدامه.

يمكن تنفيذ هذه العملية ببساطة وذلك بجعل التابع synchronized وهذا يعني:

عندما نستدعي هذا التابع فإننا نضع قفلاً على الـ object

مثال:

```
class Data
{
    private int i = 200;

    public synchronized void inc() {i++;}
    public synchronized void dec() {i--;}
    public int getI() {return i;}
}

public class Test
{
    public Data myData;

    public Test() {
        myData = new Data();
        new IncThread().start();
        new DecThread().start();
    }

    private class IncThread extends Thread
    {
        public void run() {
            for (int i = 0; i < 100; i++)
                myData.inc();
        }
    }

    private class DecThread extends Thread
    {
        public void run() {
            for (int i = 0; i < 100; i++)
                myData.dec();
        }
    }
```

```
    public static void main(String[] args) {
        Test t = new Test();
        Thread me = Thread.currentThread();
        try {
            me.sleep(1000);
        }catch (InterruptedException ex) {}

        System.out.println(t.myData.getI());
    }
}
```

نلاحظ في المثال السابق أننا استخدمنا الـ Inner classes كـ threads، وهذا استخدام ممتاز لـ Inner class، كما نلاحظ أننا استخدمنا تابعاً غريباً Thread.currentThread()، وهو تابع يعيد مؤشراً على الـ thread الذي استدعي بداخله، وبما أننا أشرنا سابقاً إلى أن البرنامج الرئيسي عبارة عن thread فإن التابع السابق أعاد مؤشراً على الـ thread الرئيسي للبرنامج والذي قمنا بتنويمه قليلاً حتى نضمن انتهاء الثريدين من عملهما في الزيادة والإنقاص، وذلك لنضمن أننا نطبع قيمة i النهائية.

ملاحظة أخيرة:

إن انتهاء البرنامج مقترن بانتهاء عمل جميع الـ threads التي أنشأها.

ومع نهاية هذه المحاضرة تكون قد انتهيت محاضرات لغات البرمجة، والتي أسأل المولى العلي القدير أن يتقبلها مني وأن ينفع بها كل من يقرؤها..

وأحب أن أشكر كل من شجعني ودعمني لإتمام هذا العمل المتواضع، وأستميحكم عذراً على كل نقص أو خطأ ورد في المحاضرات السابقة فكل من لا يخطئ..

وأريد في النهاية أن أثني على كل من حقق فريق Lectures Team ولكن ليس بأشخاصه وإنما بفكرته، لأن الأشخاص يتبدلون ولكن الفكرة تبقى، فأرجو أن تكون وفقنا لعمل نهضة حقيقية في مجال كتابة المحاضرات، وأن يبقى هذا العمل شراعاً لكل من يريد أن يدخل في هذا السلك، وباب Lectures Team مفتوح لكل من يحب أن ينزل محاضرات في المستقبل، ولكن بشرط التزام شروط العمل في الفريق.

أتمنى للجميع النجاح والتفوق في هذه المادة وفي جميع المواد..

أخوكم وزملائكم: عمار ياضر ياضر.

انتهت المحاضرة

lectures_team@hotmail.com