חישוב אסינכרוני והחזרת תוצאה

תקציר

Guarded suspension

Observer pattern

Callback function

Fluent programming

ד"ר אליהו חלסצ'י
eliahukh@colman.ac.il

במסמך זה נדמה כיצד חלק מהכלים הנתונים לרשותנו עובדים מאחורי הקלעים.

נתחיל בליצור איזושהי מחלקה בשם Caller, החושפת ממשק בשם Callable. ממשק זה מגדיר מתודה בשם ()call שיכולה להחזיר ערך V. ה Caller יפעיל את ה Callable בדרכים שונות.

כדוגמא ראשונה נבצע קריאה סינכרונית פשוטה ל ()call במתודה בשם ()syncExec.

```
public class Caller {

	public interface Callable<V>{
		V call();
	}

	public <V> V syncExec(Callable<V> c){
		return c.call();
	}
}
```

נשים לב שהמחלקה Caller אינה תלויה בטיפוס V, אלא המתודה ()syncExec היא זו שתלויה ב V.

דוגמאות לשימוש:

נגדיר את המחלקה Number

```
	public static class Number{
		private double d;
		public Number(double d) {
			this.d=d;
		}
	}
```

וכעת ניצור Caller שיריץ את היצירה שלNumber. נעבור בהדרגה ממחלקה אנונימית לקיצורים של lambda expressions:

```
	public static void main(String[] args) {
		Caller c=new Caller();
		Number n;
		// with anonymous class
		n=c.syncExec(new Callable<Number>() {
			@Override
			public Number call() {
				return new Number(5);
			}
		});
		// with a lambda expression
		n=c.syncExec(()->{return new Number(5);});
		// with a shorter lambda expression
		n=c.syncExec(()->new Number(5));
	}
```

נחזור ל Caller שלנו. נוסיף לו את המתודה ()aSyncExec שתריץ את ה ()call בת'רד נפרד. נניח לרגע שבמימוש נחזיר V כמו קודם:

```
	public <V> V aSyncExec(Callable<V> c){
		new Thread(()->c.call()).start();
		return ???
	}
```

מה יהיה ערך החזרה כרגע? הרי אין לנו מושג כמה זמן תיקח הקריאה ל call וההחזרה של V, בעוד שהקריאה ל ()start של Thread מיד חוזרת, ולכן גם ()aSyncExec מיד חוזרת; כפי שהיא אמורה להיות, שכן, ההרצה של call צריכה להיות אסינכרונית.

Guarded suspension

האופציה הראשונה שנממש, תהיה החזרה של טיפוס בשם Future<V> שיכול להכיל איזשהו משתנה מטיפוס פרמטרי V. את ה Future נוכל להחזיר מיד, בעוד שה V שלו עדיין null. בתום העבודה של call נזין ל Future את הערך V ש call החזירה.

```
public class Future<V>{
        V v;
        public void set(V v){this.v=v;}
        public V get(){return v;}
}


public <V> Future<V> aSyncExec(Callable<V> c){
        Future<V> f=new Future<>();
        new Thread(()->f.set(c.call())).start();
        return f;
}
```

בבוא העת, נוכל לגשת ל Future ולשלוף ע"י קריאה ל ()get את ה V שיצרנו. לדוגמא:

```
Future<Number> f=new Caller().aSyncExec( ()->{
                    /* really long code*/
                    return new Number(5);
                  });
// do some things here in parallel, and then
Number result=f.get();
```

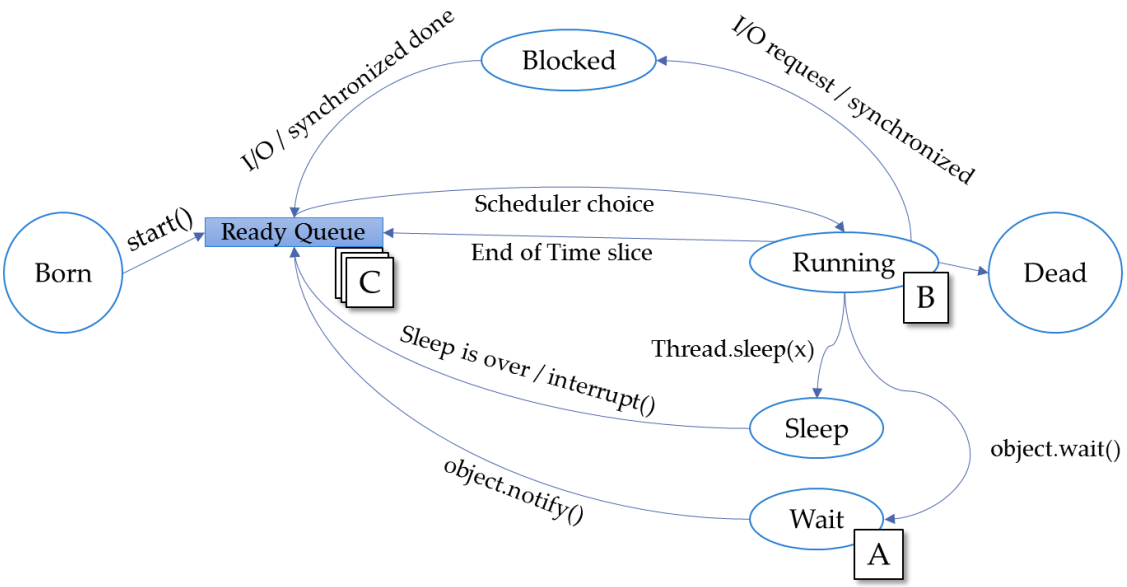
אבל רגע! מה יקרה אם הקריאה ל ()f.get תהיה לפני הזמן? אם התר'ד שמריץ את ()call לא סיים, אז ()f.get תחזיר null, כי הרי התר'ד עוד לא הפעיל את ()f.set.

כדי לפתור את הבעיה, נשתמש ב guarded suspension. במילים פשוטות, נניח שאם מישהו קרא ל ()f.get מתוך ה Future שלנו, סימן שהוא זקוק למידע של V וממילא לא יכול להמשיך בלעדיו. לכן, נגרום לו להמתין עד אשר יוזן ערך ל V. נעשה זאת באמצעות wait ו notify.

```
public class Future<V>{
        V v;
        public synchronized void set(V v){
                this.v=v;
                notifyAll();
        }
        public V get(){
                if(v==null)
                        synchronized (this) {
                                try {
                                        wait();
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                        }
                return v;
        }
}
```

תזכורת: כאשר ת'רד מסוים קורא ל wait מתוך איזשהו אובייקט, הת'רד יעבור למצב waiting. כלומר הוא אינו נמצא ב ready queue ולכן אין לו סיכוי לזכות בזמן ריצה – המתנה ללא busy waiting...

הוא יימצא שם עד שת'רד אחר יקרא ל notify מתוך אותו האובייקט. קריאה ל notifyAll תשחרר את כל הת'רדים שקראו ל wait מאותו האובייקט ונתקעו.



נשים לב לכמה פרטים במימוש שלנו ל Future.

א. קריאות ל wait או ל notify/notifyALL צריכות להיות מסונכרנות. תארו לכם מצב שיש בו כמה ת'רדים שרצים במקביל ומתבצעות קריאות ל wait ול notify. חוסר סנכרון עבור קריאות אלה יכול לפגום בלוגיקה שלנו עבור מנגנון ההשהיה וההתעוררות... לכן, ה JVM יזרוק לנו IllegalMonitorStateException במקרה זה. זהו unchecked exception, אינכם מחויבים לתפוס אותו כי אתם גם לא צריכים. עליכם פשוט להבין ולזכור את הפתרון – שימוש ב synchronized.

ב. יכולות להיות הרבה מאד גישות למתודת ה get() ע"י ת'רדים שונים לאורך כל ריצת התוכנית. לכן, (1) במתודת ה set קראנו ל notifyAll, ו (2) במתודת ה get הפעלנו synchronized רק אם ה v שווה ל null. כלומר, (1) אם היו מלא ת'רדים שבקשו את ה get לפני הזמן, כולם נתקעו על ה wait של אותו אובייקט Future. כשיופעל ה set אז notifyAll תשחרר את כולם. (2) רק עבור הת'רדים האלה נשלם את המחיר של synchronized במתודת ה get. בהמשך התוכנית ה v כבר לא null, ולכן נדלג על הקריאה ל synchronized, שכזכור, היא קריאה יקרה.

ג. מה שעלול לבלבל זו הקריאה ל wait בתוך synchronized. לכאורה, לעולם לא נצא מהבלוק של ה synchronized, ולכן האובייקט יישאר נעול, ולכן ת'רד אחר לא יוכל להריץ את מתודת ה set(), הרי גם היא מסונכרנת. אל אין כל בעיה. קריאה ל wait משחררת את הנעילה של synchronized והכל מסתדר.

נבדוק שהקוד שלנו אכן עובד ע"י דימוי של משימה ארוכה (5 שניות) שרצה במקביל:

```
Future<Number> f=new Caller().aSyncExec(()->{
        try { Thread.sleep(5000);} catch (InterruptedException e) {}
        return new Number(5);});

System.out.println("this is done in parallel");
// now my watch begins... :{
Number result=f.get(); // and now it ends  :)
System.out.println(result.d);
```

Observer Pattern

החיסרון במימוש הקודם שלנו הוא שנאלץ להמתין זמן שעלול להיות זמן רב, לתוצאת החישוב. אמנם עד לנק' בה אנו זקוקים לתוצאה עשינו דברים במקביל, אך מאותה הנק' אנו ממתנים.

במקום זאת, נוכל להשתמש ב observer pattern. פשוט נבקש עדכון כאשר תוצאת החישוב הסתיימה, וכשנקבל את העדכון, נשלוף את הערך ונשתמש בו.

נממש זאת דווקא על ה Future שיצרנו כדי להמחיש את ההבדל בין שתי הטכניקות:

```
public class Future<V> extends Observable{
        V v;
        public void set(V v){
                this.v=v;
                setChanged();
                notifyObservers();
        }
        public V get(){return v;}
}
```

וב main:

```
Future<Number> f=new Caller().aSyncExec(()->{
        try { Thread.sleep(5000);} catch (InterruptedException e) {}
        return new Number(5);});

f.addObserver(new Observer() {
        @Override
        public void update(Observable o, Object arg) {
                Number result = ((Future<Number>)o).get();
                System.out.println(result.d);
        }
});


System.out.println("this is done in parallel");
```

נשים לב לכמה דברים:

א. אין צורך בהמתנה או ב synchronized

ב. המתודה set הודיעה לכל ה observers שקרא שינוי. כלומר, טפלנו במקרה של ת'רדים רבים שתוהים מתי לבצע get() ואינם רוצים לקרוא ל get() לפני הזמן.

ג. הגדרנו מראש מה צריך לקרות לכשיגיע V. עשינו זאת ע"י ההוספה של ה observer האנונימי לעיל. הוא שולף את ה result ומדפיס אותה. בינתיים ה main שלנו יכול להמשיך במקביל בעיסוקיו השונים **מבלי לחכות לערך החזרה.**

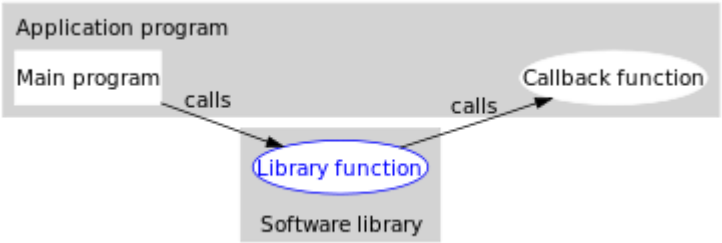
ד. השימוש במשתנה f הוא מיותר; הרי ישנה טרנזיטיביות בקריאות. בואו נקצר קצת את ה main

```
new Caller().aSyncExec(()->{
        try {Thread.sleep(5000);} catch (InterruptedException e) {}
        return new Number(5);}).addObserver( (Observable o, Object arg)->{
                Number result = ((Future<Number>)o).get();
                System.out.println(result.d);
        });
System.out.println("this is done in parallel");
```

Callback function + fluent programming

אם כבר כתבנו את הקוד לעיל, למה שלא נשתמש ב Callback? הרי זה ממילא מה שעשינו באמצעות המבנה המורכב יחסית של ה observer.

תזכורת: פונקציית callback היא קוד שעובר כפרמטר לאיזושהי פונקציה אחרת, כדי שזו תריץ אותו בסופה. כלומר, מי שיזם את הקריאה לפונקציה גם הזין לה את ה callback כדי שזו תריץ אותו לכשתסיים.



תזכורת: fluent Programming ניתן ליישם כאשר המתודה המופעלת מאובייקט מסוים מחזירה את אותו האובייקט (או אובייקט חדש מאותו הסוג) כדי שיהיה אפשר להמשיך ולשרשר מתודות.

לדוגמא (מ ++C)

```
class Num{
        int x;
        public:
        Num(int x){this->x=x;}
        Num& inc(){ x++; return *this;}
};
```

ואז ניתן לשרשר את הקריאות למתודה inc:

```
Num n(0);
n.inc().inc().inc(); // 3
```

חזרה לחישוב במקביל אצלנו. נרצה שלאחר שהמשימה שרצה במקביל תסתיים, היא תריץ פונקציית callback כלשהי שאנו נגדיר לה. בנוסף, נרצה את האפשרות להמשיך ולשרשר פונקציות callback שכולו יקרו אחת אחרי השנייה – <u>כך שהפלט של פונקציה אחת יהווה הקלט של זו שבאה אחריה.</u>

לדוגמא:

```
Callable<String> task=new Callable<String>() {
        @Override
        public String call() {
                try {Thread.sleep(5000);} catch (InterruptedException e) {}
                return "42";
        }
};

new Caller().aSyncExec(task) // we got a Future<String>
.thenApply((String s)->Integer.parseInt(s)) // we got a Future<Integer>
.thenApply((x)->2*x) // we got a Future<Integer>
.thenAccept((x)->System.out.println(x)); // we returned void

System.out.println("this is done in parallel");
```

בדוגמא לעיל יצרנו משימה (task) שלאחר 5 שניות מחזירה את המחרוזת "42". לאחר מכן הרצנו את task במקביל באמצעות ה Caller שלנו.

כעת נרצה לאפשר שרשור של פונקציות שירוצו אחת אחרי השנייה, באותו הת'רד של ה Caller, כאשר הפלט של האחת יהיה הקלט של הבאה אחריה. נעשה זאת באמצעות מתודה בשם ()thenApply.

הבה נעקוב אחר הדוגמא. לאחר ש task החזירה String, תופעל הפונקציה שמקבלת String ומחזירה את המרתה ל Integer. מייד אח"כ הגדרנו שתופעל פונקציה שבהינתן x כלשהו, תחזיר את 2*x, ולבסוף הפעלנו את המתודה ()thenAccept שהוסיפה את הפונקציה שבהינתן x כלשהו היא תדפיס אותו. נשים לב שפונקציה זו שונה מהשאר שכן, אין לה ערך חזרה.

ה main המשיך במקביל והדפיס this is done in parallel, ולאחר 5 שניות, בת'רד של ה Caller, תודפס התוצאה 84.

כיצד נעשה זאת? כיצד נממש את המתודות thenApply ו thenAccept? למי הן שייכות? נסו לפתור זאת לבד לפני שתמשיכו אל הפתרון (המופיע בעמוד הבא).

ובכן, הקריאה ל ()aSyncExec החזירה לנו מיד אובייקט מסוג Future (כאשר ה V לו יוזן מאוחר יותר).

- לפיכך, המתודה ()thenApply צריכה להיות של המחלקה Future.
- כדי לקבל פונקציה שמקבלת פרמטר מסוג אחד, ויכולה להחזיר פרמטר מסוג אחר, נצטרך לשם כך להגדיר ממשק מתאים.
- כדי שנוכל להמשיך לשרשר פעולות של ()thenApply וכן ()thenAccept נצטרך שמתודות אלו יחזירו אובייקט מסוג Future.
- נצטרך שזה לא יהיה אותו האובייקט, אלא אובייקט חדש, שכן, כל Future כזה יעבוד עם טיפוסים שונים.
  - בדוגמא לעיל, הקריאה ל ()aSyncExec החזירה Future<String>, מתוכו הפעלנו את ()thenApply שהחזירה בתורה אובייקט חדש מסוג Future<Integer>, מתוכו הפעלנו את הפונקציה הבאה שהחזירה גם היא אובייקט חדש מאותו הסוג, עד ש ()thenAccept הופעלה מאובייקט Future זה, והחזירה void שעצר את השרשור. ראו את ההערות בקוד.
- המתודה ()thenAccept תצטרך לקבל אובייקט מסוג ממשק שונה, ממשק שיגדיר מתודה שתקבל ערך V כלשהו אך תחזיר void. אם זו לא היתה מתודה שונה, היינו מקבלים שגיאת קומפילציה, שכן הקריאה ל ()System.out.println לא מחזירה דבר.

אם לא הצלחתם לחשוב על הפתרון לבד, זו עוד נקודה שכדאי לעצור בה ולנסות לממש את התיאור לעיל. הפתרון בעמוד הבא.

תחילה נגדיר לנו ממשקי עזר עבור הפונקציות של thenApply() ו thenAccept():

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

public interface Applicable<Return,Param>{Return call(Param p);}

public interface Applier<Param>{void call(Param p);}
```

הראשון הוא Callable שהגדרנו קודם, יודע להחזיר ערך V. השני מגדיר מתודה שמקבלת פרמטר מסוג Param ומחזירה ערך מסוג Return, יהיו אשר יהיו. ואילו השלישי מגדיר מתודה שמקבלת פרמטר מסוג Param אך מחזירה void.

תזכורת: המתודה aSyncExec() של Caller, מחזירה Future חדש, ומפעילה אצלו את מתודת ה set עם הערך ש call() החזירה. נשים לב ש f.set() מתבצעת בתוך הת'רד של ה Caller.

```
public <V> Future<V> aSyncExec(Callable<V> c){
	Future<V> f=new Future<>();
	new Thread(()->f.set(c.call())).start();
	return f;
}
```

כעת ניגש למחלקה Future ונעדכן אותה בהתאם:

```
public class Future<V>{
	V v;
	Runnable r;
	public void set(V v){this.v=v; r.run();}

	public <Return> Future<Return> thenApply(Applicable<Return, V> app){
		Future<Return> f=new Future<>();
		r=()->f.set(app.call(v));
		return f;
	}

	public void thenAccept(Applier<V> app){
		r=()->app.call(v);
	}
}
```

נתחיל ממתודת ה set. אנו רואים שהיא מזינה את v ולאחר מכן מפעילה את run() של השדה Runnable r. האתחול של r נעשה במתודות thenApply ו thenAccept.

נתבונן ב thenApply, היא מקבלת אובייקט (app) מסוג Applicable עם Return ו V. כלומר, app היא פונקציה שחייבת לקבל משתנה מסוג V כפרמטר, ויכולה להחזיר ערך מסוג כלשהו Return.

בפנים היא יוצרת Future חדש שה V שלו הוא Return; כלומר מאותו הסוג ש app מחזיר. בשורה הבאה אנו מגדירים ל r להצביע על Runnable חדש, שבתורו מזין ל future שיצרנו את ה Return שחוזר מה call של app בהינתן הערך v. ולאחר מכן אנו מחזירים את ה f החדש שיצרנו.

בואו נעקוב אחר תחילת הדוגמא שלנו:

```
new Caller().aSyncExec(task) // we got a Future<String>
.thenApply((String s)->Integer.parseInt(s)) // we got a Future<Integer>
```

עבור דוגמא זו, thenApply הופעלה מתוך Future<String>, כלומר v שלו הוא String. היא קבלה את app שעבורו ה Param הוא לא אחר מאשר V, כלומר String, והחזירה Integer. לפיכך Return הוא Integer. לכן, כשיצרנו את ה f <Future<Return החדש היה זה Future<Integer>. והשרשור של המתודות הבאות של thenApply או thenAccept יהיה כבר ממנו.

כמו כן, המתודה thenApply הגדירה כבר כעת ש r יפעיל את app.call(v) כאשר יריצו אותו, כלומר כאשר יקראו ל r.run(). ניתן לראות שקריאה זו ל r.run() מתבצעת בתוך מתודת ה set רק לאחר שהוזן הערך ל v. ומכיוון שהת'רד של Caller הפעיל את ה set הזו, אז גם הוא זה שהפעיל את r.run(). כלומר, בדוגמא שלנו הפיכת המחרוזת ל Integer בוצעה באותו הת'רד של ה Caller.

כאמור, thenApply החזירה מיד Future חדש, אך ה V שלו יוזן רק כש r.run תרוץ, שכן היא קוראת ל f.set(app.call(v)); כך, בהדרגה, ה set הזו מתוך ה Future החדש תפעיל בתורה את הפונקציה הבאה, וחוזר חלילה. לכן, בכל פעם שנשרשר קריאה של thenApply, הן תקראנה אחת אחרי השנייה, מתוך אובייקטים שונים של Future, כאשר הקלט של הפונקציה האחת הוא הפלט של קודמה.

מתישהו נרצה לעצור את ההגדרות של שרשור הפונקציות, נעשה זאת באמצעות thenAccept שפעולת באופן דומה, פרט לעובדה שאינה מחזירה Future חדש, אלא פשוט void.

מה שראינו במסמך זה, זו הדמיה חלקית ביותר של מה שנעשה מאחורי הקלעים בספריות הקוד של java.util.concurrent. כמובן, שם יש קוד הרבה יותר מורכב ועשיר בדרכים שונות באופן שגם נותן מענה ללא מעט מקרי קצה. אך כאן יצרנו כלים דומים שממחישים את אופן פעולתם של מנגנונים אלה.

כעת עליכם לקרוא על הכלים האמתיים. אנא קראו אודות Callable, Future, CompletableFuture.

בהצלחה!

אלי.