Lambda Expression

"A closures by any other name will function all the same"

“แม้จะบิดบังใช้ชื่ออื่นๆ แต่การทำงานก็ยังเหมือนเดิม”

Shakespeare

เขียนโปรแกรมเชิงฟังก์ชัน

ก่อนจะเป็นประพจน์ Lambda (แลมบ์ดา) ความเป็นมาเกิดจาก แนวคิดการเขียนโปรแกรมเชิงฟังก์ชัน คำว่าเชิงฟังก์ชัน (Functional) เป็นคำที่ใช้ในความหมายต่างๆ กัน ในโลกการเขียนโปรแกรม การเขียนโปรแกรมเชิงฟังก์ชัน หมายถึง การคิดเกี่ยวกับปัญหาเชิงโดเมน (Domain) ที่ค่าแปรผันไม่ได้ (immutable) และฟังก์ชันใช้ทำงานกับค่าเหล่านั้น

ลองพิจารณา ฟังก์ชัน การคำนวณด้านทั้งสามด้านที่สัมพันธ์กันของรูปสามเหลี่ยม พิธาโกรัส

$x^2 + y^2 = z^2$

สูตรคำนวณนี้เป็นเป็นหาเชิงโดเมน ที่เกี่ยวกับเรื่องด้านต่างๆ ของรูปสามเหลี่ยม และด้านแต่ละด้านของ x และ y มีค่าแน่นอน จึงคำนวณค่า z ได้ คงเป็นไปไม่ได้ ที่จะคำนวณค่า z ได้ โดยที่จะเปลี่ยนให้ ค่า x หรือ y เป็น y++ หรือ x++ แต่จะทำได้ก็ต่อเมื่อคำนวณให้เสร็จก่อน แล้วค่อยกำหนดค่าไปใหม่ นี้คือลักษณะที่เรียกว่า แปรผันระหว่างคำนวณไม่ได้

การเขียนโปรแกรมเชิงฟังก์ชันมีลักษณะเป็นการประกาศว่าอยากได้อะไร (declarative) มากกว่าที่จะบอกว่าทำงานอย่างไร (imperative) เช่น การเขียนสูตร พิธาโกรัส อ่านแล้วเข้าใจได้เลยว่า มาจากอะไร แต่ถ้าเราเขียนใหม่ว่า

```
int x, y, z;
int _x = x * x;
int _y = y * y:
z = _x + _y
```

ซึ่งเป็นแสดงวิธีการหาค่าที่ได้มา ที่ละขั้นตอน อย่างนี้ไม่ถือเป็นการเขียนโปรแกรมเชิงฟังก์ชัน เมื่อดูอย่างนี้ การเขียนให้เป็นฟังก์ชันจะอ่าน แล้วเข้าใจได้มากกว่า นี้คือข้อดีอย่างหนึ่งของการเขียนโปรแกรมเชิงฟังก์ชัน

แนวคิดการเขียนโปรแกรมเชิงฟังก์ชันนี้ พัฒนามาจากแนวคิดทางคณิตศาสตร์ ในปี 1930 กว่าๆ Alonzo Church ได้พัฒนา Lambda Calculus ซึ่งเป็นรูปแบบหนึ่งที่เรียกใช้งานเป็นรูปฟังก์ชัน และรูปแบบนี้ก็นำมาใช้เป็นรูปแบบในการเขียนโปรแกรมเชิงฟังก์ชัน และคำว่าแลมบ์ดา นี้ จึงเป็นความหมายที่แทนฟังก์ชันได้

มีภาษาสมัยใหม่หลายตัวที่สนับสนุนการเขียนโปรแกรมในลักษณะฟังก์ชัน เช่น C#, Groovy, Ruby, Python, Scalar ซึ่งล้วนเป็นภาษาที่ได้รับความนิยมสูง Java เองก็ไม่อยากจะตกขบวนไปกับเขา เราใช้การเขียนแบบฟังก์ชันผสมกับการเขียนโปรแกรมเชิงวัตถุได้ (ฟังก์ชันของเชิงวัตถุเรียกเมธอด ซึ่งเป็นส่วนหนึ่งของคลาส)

ก่อนมีประพจน์แลมบ์ดา (Lambda Expression)

ก่อนมี Java 8 เริ่มมีการเขียนเชิงฟังก์ชันบางแล้วแล้ว แต่ยังไม่อยู่ในรูปแบบแลมบ์ดา ส่งตัวแปรเข้าแบบกระชัดและรวบรัด ลองมาดูการเขียนเชิงฟังก์ชันแรกกัน ที่ใช้งานในการลงทะเบียนเหตุการณ์ให้ปุ่มคำสั่ง การส่งตัวแปรเข้าในรูปคลาสไม่มีชื่อ หรือ เรียกว่า anonymous class และคลาสไม่มีชื่อนี้ มีฟังก์ชันที่เขียนขึ้นทันที่ว่าให้ทำอะไร การส่งตัวแปรแบบนี้ถือว่าเป็นการส่งแปรเข้าฟังก์ชันอย่างหนึ่ง และต่อไปจะลดจากคลาสไม่มีชื่อเป็น เพียง ฟังก์ชันไม่มีชื่อ ซึ่งกลายเป็นประพจน์แลมบ์ดา ที่สมบูรณ์แบบ

ตัวอย่าง `1. a functional programming`

```
import java.awt.event.ActionEvent;
```

```
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JButton;

public class FirstLamda {
    public static void main(String[] args){
        JFrame frame = new JFrame("My Frame");
        JButton button = new JButton("OK");
        //using anonymous inner class as an argument of button's function
        button.addActionListener( new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button clicked");
            }
        });
        frame.add(button);
        frame.setSize(200,200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

ความหมาย Lambda Expression

การเปลี่ยนแปลงที่สำคัญของ Java8 การเขียนให้มีประพจน์แลมบ์ดา ได้ คำว่าแลมบ์ดา มาจากอักษรภาษากรีก **λ** (อ่านว่า แลมบ์ดา) มีความหมายตามภาษานี้ว่า ฟังก์ชันที่ยังไม่กำหนดหน้าที่ทำงานไว้ หรือตามภาษาเขียนโปรแกรม ดูคล้ายเป็น abstract method ที่อยู่ใน abstract class ในภาษา Java แลมบ์ดา หมายถึง การใช้ฟังก์ชันที่ยังไม่มีชื่อ เป็นตัวแปรเข้าหรือตัวแปรออก จากฟังก์ชันอีกที ฟังก์ชันไม่มีชื่อนี้ใช้ไม่ได้กับ abstract method เพราะ แลมบ์ดา มีไทป์(type) เป็น functional interface และมีเพียงเมธอดเดียว

ตัวอย่าง 2. abstract class

```
public abstract class AbstractJob {
    abstract void doJob();
}
```

ตัวอย่าง 3. การประกาศ FunctionInterface

```
@FunctionalInterface
public interface IJob {
    public void job();
}
```

การประกาศ @FunctionalInterface ก็เพื่อแจ้งให้คอมไพล์เลอร์รู้ว่า เป็นชนิด Functional Interface ซึ่งหมายความว่า จะเป็นอินเทอร์เฟส ที่มีเพียงเมธอดเดียว แต่ถ้ามีหลายตัว หลายตัวเหล่านั้นจะต้องเป็นเมธอดใน คลาส Object การทำแบบนี้จะถือว่าเป็นการประกาศใหม่ (Override)

แลมบ์ดา ใช้กับกำหนดฟังก์ชันในอินเทอร์เฟส ตามที่ทราบกันดีแล้ว อินเทอร์เฟส นั้นตัวมันเองไม่สามารถใช้งานได้โดยตรง การใช้งานต้องผ่านคลาส ต่อยอดงานจาก อินเทอร์เฟส ซึ่งหมายความว่าจะต้องเขียน เนื้องานของฟังก์ชันของอินเทอร์เฟสนั้นอีกที

ตัวอย่าง 4. การนำอินเทอร์เฟส IJob มาใช้งานในคลาส MyJob

```
public class MyJob implements IJob {
     @Override
```

```
     public void job(){
       System.out.println("Functional Programming");
     }
}
```

การใช้แลมบ์ดา ใช้งานในลักษณะเดียวกับคลาสที่สืบทอดงานจากอินเทอร์เฟส แต่ไม่ต้องประกาศชื่อของฟังก์ชันของอินเทอร์เฟสนั้น (ดูตัวอย่างที่ 5) ประกาศชนิดเป็นเพียงชื่ออินเทอร์เฟส การประกาศชนิดหรือไทป์นี้เอง จึงเป็นเหตุผลว่าแลมบ์ดา มีไทป์เป็น ฟังก์ชันนอนอินเทอร์เฟส และทำการเขียนโปรแกรมประกาศการทำงานได้เลย ซึ่งจะเห็นว่า เราไม่จำเป็นต้องสร้างคลาสเพื่อสืบทอดจากอินเทอร์เฟส แล้วทำเป็นวัตถุจากคลาสที่สืบทอด ผลคือสั้นกว่ามาก และที่สำคัญสะดวกใช้งาน

ตัวอย่าง 5. การประยุกต์ใช้แลมบ์ดา

```
public static void main(String[ ] agrs) {
    IJob myJob = ( ) -> System.out.println("Functional Programming ");
    myJob.job( );    // print Functional Programming
}
```

จากตัวอย่างที่ผ่านมาการใช้แลมบ์ดา จะต้องเรียกใช้งานชื่อของฟังก์ชันในอินเทอร์เฟส (myJob.job( )) แต่ตอนสร้างหรือกำหนดรายละเอียดการทำงาน ไม่ต้องระบุชื่อ เพราะ จาวา จะรู้จากสภาพแวดล้อมแล้วว่า หมายถึง ฟังก์ชัน job ใน IJob ซึ่งมีเพียงฟังก์ชัน job( ) ตัวเดียว

ประโยชน์การใช้งานแลมบ์ดา

แล้วอะไรคือประโยชน์ของการใช้แลมบ์ดา ประโยชน์ที่เห็นได้ชัด คือ เมื่ออยากใช้งานเมื่อใดก็ประกาศการใช้งานได้เลย ไม่ต้องระบุตายตัวจากสร้างออปเจค ลองดูตัวอย่างที่ 6 มีการใช้งานได้สองลักษณะ ทำงานไม่เหมือนกัน

ตัวอย่าง 6. การใช้แลมบ์ดาเดิมหลายครั้ง

```
public static void main(String[ ] agrs) {
    IJob myJob = ( ) -> System.out.println("Functional Programming");
    myJob.job( ); //Print: Functional Programming
    IJob yourJob = ( ) ->System.out.println("Lambda Expressing");
    yourJob.job( ); //Print: Lambda Expressing
}
```

อีกครั้ง ถ้าเราใช้ anonymous class แทนการใช้แลมบ์ดา (ตอนนี้เรียกแลมบ์ดา ว่าเป็น anonymous function) ก็ใช้งานได้เหมือนกัน แต่ลองดูว่า การเขียนโปรแกรมแบบไหนยุ่งยากกว่ากัน เทียบกับ ตัวอย่างก่อนหน้านี้

ตัวอย่าง 7. การสร้าง anonymous class

```
public static void main(String[ ] agrs) {
    AbstractJob myJob2 = new AbstractJob (){
        @Override
        void doJob(){
                System.out.println("Functional Programming again");;
        }
    };

    myJob2.doJob();//print: Functional Programming again
}
```

การอ้างตัวแปรของแลมบ์ดา เป็น final

มีสิ่งหนึ่งที่ควรระวัง ตัวแปรของฟังก์ชันต้องเป็นค่าสุดท้าย final ซึ่งหมายความว่า เราจะแก้ไขค่านี้ในฟังก์ชันไม่ได้ ขณะฟังก์ชันทำงาน ใน Java 8 เข้าใจว่า ต้องส่งเป็นค่าง final อยู่แล้ว จึงไม่จำเป็นต้องระบุว่า final ก็ได้

ตัวอย่าง 7. การส่งค่า final ไปยังฟังก์ชัน

```
public static void main(String[ ] agrs) {
    final String job = "Programmer";
    IJob myJob = ( ) -> {
                              String moreJob = "Writer";
                              System.out.println(job + "," + moreJob);
                         };
    myJob.job( );    // print Programmer, Writer
}
```

การอ้างอิงตัวนี้ ใน แลมบ์ด้า เรียกว่า **โคสเซอร์ (Closures)** ความไม่เหมือนกันของ[1] แลมบ์ด้า และ โคสเซอร์ กล่าวคือ แลมบ์ด้า จะหมายถึง การส่งตัวแปรเข้า หรือออก ในรูปฟังก์ชัน ในขณะที่ โคสเซอร์ จะหมายถึงการที่ฟังก์ชันอ้างตัวแปร ที่อยู่ของนอกพื้นที่อ้างอิงของฟังก์ชัน ที่ไม่ส่งแบบตัวแปร หรือสร้างเองในฟังก์ชัน ตัวแปรอ้างอิงนี้ สำหรับจาวา มีข้อจำกัด ให้ต้องเป็นค่า final เท่านั้น

การเลี่ยงค่า final จะทำอย่างไร ถ้าเราต้องการให้ค่านี้มีการเปลี่ยนแปลงได้ วิธีหนึ่งต้องแก้ให้เขียนโปรแกรม แบบ inner class/inner interface ลองดูจากตัวอย่างต่อไปนี้ จะเห็นว่า โปรแกรมทำงานได้ ตัวแปร job ถูกดัดแปลงภายในคำสั่งแลมบ์ดา ได้

ตัวอย่าง 8. อินเทอร์เฟสภายในคลาส

```
public class Main {
    public interface IJob {
        void job();
    }
    static String job = "Programmer";
    public static void main(String[] agrs){
        IJob myJob = () -> {
            job = "Writer";
            System.out.println(job);
        };
        myJob.job();
    }
}
```

ไวยากรณ์ของคำสั่งแลมบ์ดา

ประโยคคำสั่งแลมบ์ดา ประกอบด้วย รายการตัวแปร ซึ่งอยู่ในวงเล็บ และส่วนเนื้อหา อยู่ ใน ปีกกา บางที่ถ้ามีคำสั่งเดียวไม่ต้องมีปีกกาก็ได้ และใช้ลูกศร (->) คั่นกลางระหว่าง รายการตัวแปร กับส่วนเนื้อหาการทำงาน โดยรวมเหมือนการประกาศเมธอด สำหรับส่วนที่ต่างกันคือ แลมบ์ดา ไม่ชื่อฟังก์ชัน และไม่จำเป็นต้องประกาศชนิดข้อมูลรายการตัวแปร แต่จะใส่ก็ได้ (โดยมากเขาไม่ใส่กัน) ตารางต่อไปนี้เปรียบเทียบการเขียนแบบเมธอดกับการใช้แลมบ์ด้า เพื่อความเข้าใจในไวกรณ์หรือสำนวนของฟังก์ชันนอนโปรแกรมมิ่ง

[1] Eric Elliott, “Programming JavaScript Applications”, O’Reilly Media, Inc., 2014, (Page 17).

ตาราง 1. การเปรียบเทียบระหว่าง การใช้คำสั่งแลมบ์ดา และใช้แบบเมธอด

Lambda Expression	Method

`( ) -> { System.out.println("No job");}`	`void println( ) {`
	`  System.out.println("No job");`
	`}`

`(a) ->{ System.out.println(++a); };`	`void increase( int a ){`
	`  System.out.println(++a);`
	`}`

`a -> System.out.println(++a);`	`void increase( int a ){`
	`  System.out.println(++a);`
	`}`

`(a, b) ->{ if (a > b) return a; else return b;};`	`int greater( int a, int b ) {`
	`  if( a>b ) retrun a; else return b;`
	`}`

`(s) ->System.out.println(s) ;`	`void printString(final String s) {`
	`  System.out.println(s);`
	`}`

จากตัวอย่างการใช้ในตาราง 1 เรามาทดสอบกันว่า มีการใช้งานอย่างไร แลมบ์ดา แรก ไม่มีตัวแปรเข้า มีการคืนค่าเป็น void ดูตัวอย่างได้จาก interface F1 แลมบ์ดา ตัวที่สอง มีตัวแปรเข้าหนึ่งตัว มีการคืนค่าเป็น void แทนด้วย interface F2 และ แลมบ์ดาที่สาม มีตัวแปรเข้าสองตัว มีชนิดข้อมูลเป็น int และมีการคืนค่าเป็น int แทนด้วย interface F3 และมีการคืนค่าเป็น int แสดงได้ดังตัวอย่างต่อไปนี้

ตัวอย่าง 9.

```
@FunctionalInterface
interface F1{
    void println();
}
@FunctionalInterface
interface F2{
    void increase(int a);
}
@FunctionalInterface
interface F3{
    int greater(int a, int b);
}

public class Main {
    static String job = "Programmer";
    public static void main(String[] agrs){
        F1 f1 = ()->System.out.println("I println.");
        f1.println();

        F2 f2 = (a)->System.out.println(a++);
        f2.increase(5);

        F3 f3 = (a, b)-> (a>b) ? a : b;
        int a = f3.greater(2, 3);
        println(a);
    }

    static void println(Object obj){
        System.out.println(obj.toString());
    }
}
```

Functional Interface ใน java.util.function

ในเพ็จเก็จของ java.util.function.* (การใช้เครื่องหมาย * หมายถึงเอาทั้งหมดของห้อง function ซึ่งถ้าไม่ใส่เครื่องหมายดอกจัน จะต้องระบุชื่อฟังก์ชันนอนอินเทอร์เฟสเอง ตามที่ต้องการเรียกใช้งาน) มี functional interface มากมายให้ใช้งาน ซึ่งเราไม่จำเป็นต้องเขียนขึ้นเอง ดังที่เคยทำมาก่อนหน้านี้

ตาราง 2. Functional interface ที่สำคัญ ใน Java.util

ชื่อ	**เมธอด**	**คำอธิบาย**
Function<T, R>	R apply(T t)	ใช้เมธอด apply มีตัว R เป็นตัวออก และ T เป็นตัวแปรเข้า
BiFunction<T, U, R>	R apply(T t, U u)	ใช้แบบเดียวกับ Function แต่ดำเนินการสามตัวแปร
Predicate<T>	boolean test(T t)	ใช้ทดสอบ ค่า 1 ค่า ซึ่งจะคือค่า boolean
BiPredicate<T, U>	boolean test(T t, U u)	ใช้ทดสอบ ค่า 2 ค่า ซึ่งจะคือค่า boolean
Supplier<T>	T get( )	ใช้อ่านค่าบางอย่าง
Consumer<T>	void accept(T t)	ใช้ดำเนินการค่า t ให้ทำบางอย่าง ไม่คืนค่าอะไรออกมา

แนวทางการใช้งาน ของอินเทอร์เฟส เหล่านี้ก็คล้ายๆ กับที่เราเคยได้ประดิษฐ์ functional interface เอง ลองดูจากตัวอย่างต่อไปนี้

ตัวอย่าง 10.

```
Function<Integer, Integer> invert = (i)-> i*-1;
BiFunction<Integer, Integer, Integer> plus = (i,j)->i+j;
Predicate<Integer> positive = (i)->i>0?true:false;
Supplier<String> greeting = ()->"Hello!";
Consumer<String> print = (s)->System.out.println(s);

System.out.println(invert.apply(1));//print -1
System.out.println(plus.apply(2, 3));//print 5
System.out.println(positive.test(1));//print true

print.accept(greeting.get());//print Hello!
```

จากตัวอย่างนี้ ฟังก์ชันแรก (Function) มี ค่าเจนเนอร์ริก สองตัว คือ Integer และ Integer (ทั้งสองตัวนี้เป็นคลาสเท่านั้น เช่น คลาส Customer แต่ในที่นี้เป็นคลาส Integer) ซึ่งแทน T และ R และฟังก์ชันนี้มีเมธอดชื่อ apply(T t) มีตัวแปรข้าวของเมธอดนี้หนึ่งตัว เป็น t มีไทป์เป็น Integer ตามที่ประกาศ และส่งค่าคืนเป็น R และมีไทป์เป็น Integer

ยังมี Functional Inteface ตัวอื่นอีก ที่เป็นค่าปริยาย และค่าสเตติก คือ

- default<V> Function<T, V> andThen(Function<? super R, ? extends V> after)
- default<V> Function<V, R> compose(Function<? super V, ? extends T> before)
- static <T> Function<T, T> identity( )

ต่อไปมาดูแนวทาง การใช้ สมมุติให้มีการ กลับเลขบวก-ลบ แล้วต่อมาให้ คูณยกกำลังตัวเอง ในที่นี้จึงใช้ แบบ default<V> Funciton<T, V> andThen( )

ตัวอย่าง 11.

```
Function<Integer, Integer> invert = (i)-> i*-1;
Function<Integer, Integer> doubleIt = (i)->i*i;

Function<Integer, Integer> invertDoubleIt = invert.andThen(doubleIt);
System.out.println(invertDoubleIt.apply(2));
```

นี้เป็นตัวอย่างของ Function ตัวอื่นๆ ก็มีอีก เช่น Predicate<T> ก็มีค่า default และ static เหมือนกัน ถึงตอนนี้ก็ไม่ยากเกินไปที่จะศึกษาได้ด้วยตนเอง ได้ที่เว็บของ Java ได้

สรุป

การเขียนโปรแกรมเชิงฟังก์ชัน หรือ Lambda Expression มีจุดสำคัญคือ จะต้องรู้ว่า มีตัวแปรที่ต้องใช้กี่ตัว ตัวใดเป็นตัวแปรที่ใช้ดำเนินการ และตัวแปรใดที่ใช้คืนค่า ตัวแปรที่ใช้ดำเนินการ จะใช้ใส่ในวงเล็บ ส่วนตัวแปรที่ใช้คืนค่า จะอยู่หลังเครื่องหมายลูกศร เช่น (a, b) -> (a>b)?a:b; ฟังก์ชันที่ทำเป็นเป็นอินเตอรเฟสนี้จะมีได้หนึ่งเมธอดเท่านั้น ยกเว้นจะเขียนทับในคลาส Object และนอกจากเขียนเองแล้ว Java จะมี functional interface ให้ใช้ในกลุ่ม java.util.* แนวทางการใช้งานก็เหมือนกับที่เราสร้างขึ้นเอง

อ้างอิง

[1] Sharan, Kishori,(2014) “Beginning Java 8 Language Feature”, Apress.
[2] https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html (2016-3-16)

แบบฝึกหัด

1. จงแปลง โปรแกรมต่อไปนี้ให้อยู่ของรูปการเขียนคำสั่งแบบ แลมบ์ดา

```
button. addActionListener(new ActionListener() {

                public void actionPerformed(ActionEvent event) {
                       System. out. println("button clicked" );
                }
        });
```

2. จงสร้างฟังก์ชัน จาก functional interface ต่อไปนี้ โดย F1 มีฟังก์ชัน ที่หาค่าสูงสุดของตัวแปรเข้าทั้งสามตัว ในขณะที่ F2 มีฟังก์ชัน ที่ใช้สำหรับหาค่า ต่ำสุด ของสามตัวแปรเข้า

```
@FunctionalInterface
interface Find{
    int max(int a, int b, int c);
}
@FunctionalInterface
interface Get{
    int min(int a, int b, int c);
}
```

3. จงเขียน แลมบ์ดา ให้มีการดำเนินการสามอย่างต่อเนื่องกัน เริ่มจาก นำตัวเลขสองตัวมาบวกกัน ต่อมาใช้ andThen นำผลลัพธ์นั้นมายกกำลังสอง ต่อมาใช้ andThen นำผลลัพธ์นั้นมาหารสอง
4. จงสร้างฟังก์ชัน จาก รายกำหนดต่อไป

a. `Function<Integer, Integer>`
b. `Function<Integer, String>`
c. `Function<Boolean, Boolean>`

5. จงยกตัวอย่างการเขียนโปรแกรมแบบ การสร้างแลมบ์ดา แบบคลาสไม่มีชื่อ มาเป็น การใช้งานแบบคลาสในคลาส