Lab 01 : From Math to Code (λ)

	Student ID	Name
1	67070159	วราภรณ์ สุขม่วง

Objectives:

1. (CLO3) สามารถอธิบาย และเขียนโปรแกรมตามหลักการในการคำนวณทางคณิตศาสตร์แบบ Lambda Calculus

From Math to Code (λ)

Lambda Calculus คือระบบทางคณิตศาสตร์ที่ใช้ศึกษา "การคำนวณ" โดยมองทุกอย่างเป็นฟังก์ชัน

- Syntax: $\lambda x . x + 1$
- **λ** = บอกว่าเป็นฟังก์ชัน
- x = พารามิเตอร์ (Input)
- . = ตัวคั่น
- $x + 1$ = สิ่งที่ทำ (Body of function)

การแปลง Syntax เป็น Scala

- ใน Scala ใช้สัญลักษณ์ => (Rocket symbol) แทนจุด .
- Math: $\lambda x . x + 1$
- Scala: (x: Int) => x + 1

Anonymous Functions (ฟังก์ชันนิรนาม)

ใน OOP ปกติ ฟังก์ชันต้องมีชื่อและสิงอยู่กับ Class (เรียกว่า Method) แต่ใน FP ฟังก์ชันไม่จำเป็นต้องมีชื่อ โดยเปรียบเทียบได้ดังนี้

Named Function (Method)	Anonymous Function (Lambda)
def addOne(x: Int): Int = x + 1	(x: Int) => x + 1

Note: Anonymous Functions มักใช้แบบ "ใช้แล้วทิ้ง" หรือส่งให้ฟังก์ชันอื่นทันที

- **Note สำหรับการสร้างโปรเจคใหม่**
  - สร้าง folder สำหรับวางไฟล์งาน ด้วยคำสั่ง *cd ชื่อfolder*
  - พิมพ์ ว่า *sbt new scala/scala3.g8* เพื่อสร้างโปรเจคจาก template
  - ตั้งชื่อว่า *xxxx* (warning : อาจจะต้องรอนาน)
  - พิมพ์ ว่า *cd xxxx*
  - พิมพ์ ว่า *sbt run*

Exercise 1: สร้างฟังก์ชันพื้นฐาน

กำหนดให้ $f(x) = x + 1$ และ $g(x) = x^2$
ฟังก์ชันใน Scala (ใช้ val และ Lambda syntax =>)

ทดสอบดังต่อไปนี้

val f: Int => Int = ???

//หรือ val f = (x: Int) => ???

val g: Int => Int = ???

//หรือ val g = (x: Int) => ???

println(f(5)) // ทดสอบ โดยคาดการณ์ว่าจะได้ผลลัพธ์ คือ 6

println(g(5)) // ทดสอบ โดยคาดการณ์ว่าจะได้ผลลัพธ์ คือ 25

```
import exercise.*

@main def main() : Unit =  {
  println("----------------")
  println("Exercise 1")
  ex01()    // How to use function from another folder
  println("----------------")
  println("Exercise 2")
  ex02()
  println("----------------")
  println("Exercise 3")
  ex03()
  println("----------------")
  println("Exercise 4")
  ex04()
  println("----------------")
}
```

```
def ex01(): Unit =
  val f = (x:Int) => x + 1
  val g = (x:Int) => x * x

  println(f(5))
  println(g(5))
```

```
Exercise 1
6
25
```

Exercise 2: Higher-Order Functions

สร้างฟังก์ชันชื่อ applyTwice ที่รับพารามิเตอร์ 2 ตัว:

1. ฟังก์ชัน h (ที่รับ Int และคืนค่า Int)
2. ค่าตัวเลข x (Int)

นำฟังก์ชัน h ไปกระทำกับ x สองรอบ หรือเขียนเป็นคณิตศาสตร์คือ $h(h(x))$

ทดสอบดังต่อไปนี้

```
//Implement applyTwice
def applyTwice(h: Int => Int, x: Int): Int = {
???
}
println(applyTwice(f, 5)) // ทดสอบ โดยคาดการณ์ผลลัพธ์ว่า f(5)=6, f(6)=7 -> Expected: 7
println(applyTwice(g, 5)) // ทดสอบ โดยคาดการณ์ผลลัพธ์ว่า g(5)=25, g(25)=625 -> Expected: 625
```

ตัวอย่างนี้แสดงว่า Higher-Order Function โดยสังเกตที่ type h: Int => Int คือ การบอก Scala ว่า "ขอรับพารามิเตอร์ที่เป็นฟังก์ชัน" ไม่ใช่รับค่า Int ธรรมดา

```
import exercise.*

@main def main() : Unit =  {
  println("---------------")
  println("Exercise 1")
  ex01()    // How to use function from another folder
  println("---------------")
  println("Exercise 2")
  ex02()
  println("---------------")
  println("Exercise 3")
  ex03()
  println("---------------")
  println("Exercise 4")
  ex04()
  println("---------------")
}
```

```
def ex02(): Unit =
    val f = (x:Int) => x + 1
    val g = (x:Int) => x * x

    def applyTwice(h: Int => Int, x: Int): Int = {
        h(h(x))
    }
    println(applyTwice(f,5))
    println(applyTwice(g,5))
```

```
Exercise 2
7
625
```

Exercise 3: Function Composition

การประกอบฟังก์ชัน (Composition) คือการนำผลลัพธ์ของฟังก์ชันหนึ่ง ไปเป็น Input ของอีกฟังก์ชันหนึ่ง ในทางคณิตศาสตร์: $(f \circ g)(x) = f(g(x))$

สร้างฟังก์ชัน myCompose ที่รับฟังก์ชัน 2 ตัว (func1 และ func2) และ คืนค่ากลับมาเป็นฟังก์ชันใหม่

ทดสอบดังต่อไปนี้

```
//Implement myCompose
//กำหนดให้มี Inputของ ฟังก์ชัน 2 ตัว ( func1และ  func2)
// คาดหวัง Outputคือ ฟังก์ชันใหม่ 1 ตัว ที่เมื่อรับค่า x จะทำ func2 ก่อน แล้วค่อยทำ func1
def myCompose(func1: Int => Int, func2: Int => Int): Int => Int = {
  // คืนค่าเป็น Lambda (x: Int) => ...
  ???
}
// Test
val f_of_g = myCompose(f, g) // สร้างฟังก์ชันใหม่ที่เท่ากับ f(g(x)) หรือ (x^2) + 1
println(f_of_g(5))
// Step: g(5) = 25 -> f(25) = 26
// Expected: 26
```

Output คือ $f(g(x))$ เป็น Anonymous Function ก้อนใหม่ ที่รอรับค่า x ในอนาคต

เมื่อเราเรียก val f_of_g = myCompose(f, g) โค้ดยังไม่ถูกคำนวณค่าตัวเลข แต่เป็นการ "สร้างท่อ" เชื่อมฟังก์ชันรอไว้ (Lazy definition)

```
import exercise.*

@main def main() : Unit =  {
  println("---------------")
  println("Exercise 1")
  ex01()    // How to use function from another folder
  println("---------------")
  println("Exercise 2")
  ex02()
  println("---------------")
  println("Exercise 3")
  ex03()
  println("---------------")
  println("Exercise 4")
  ex04()
  println("---------------")
}
```

```
def ex03(): Unit =
  val f = (x:Int) => x + 1
  val g = (x:Int) => x * x

  def myCompose(func1: Int => Int, func2: Int => Int): Int => Int = {
    (x:Int) => func1(func2(x))
  }
  val f_of_g = myCompose(f, g)
  println(f_of_g(5))
```

```
Exercise 3
26
```

Exercise 4: Scala Built-in Composition

ใน Scala ไม่ต้องเขียน myCompose เอง เพราะใน Function Class มี method ชื่อ .compose และ .andThen มาให้อยู่แล้ว

ทดสอบดังต่อไปนี้

val builtInCompose = f.compose(g) // เหมือน f(g(x)) -> ทำหลังไปหน้า

val builtInAndThen = f.andThen(g) // เหมือน g(f(x)) -> ทำหน้าไปหลัง (Natural reading)

println(builtInCompose(5)) // (5^2) + 1 = 26

println(builtInAndThen(5)) // (5+1)^2 = 36

```
import exercise.*

@main def main() : Unit =  {
  println("---------------")
  println("Exercise 1")
  ex01()    // How to use function from another folder
  println("---------------")
  println("Exercise 2")
  ex02()
  println("---------------")
  println("Exercise 3")
  ex03()
  println("---------------")
  println("Exercise 4")
  ex04()
  println("---------------")
}
```

```
def ex04(): Unit =
  val f = (x:Int) => x + 1
  val g = (x:Int) => x * x
  val builtInCompose = f.compose(g)
  val builtInAndThen = f.andThen(g)

  println(builtInCompose(5)) // (5^2) + 1 = 26
  println(builtInAndThen(5)) // (5+1)^2 = 36
```

```
Exercise 4
26
36
```