เขียนโปรแกรมภาษา Kotlin

สำหรับผู้เริ่มต้น

Null Safety

โดยทั่วไปตัวแปรในภาษา Kotlin นั้นไม่สามารถกำหนดค่า Null ได้ (Non-Nullable) เพื่อลดปัญหาการเกิดสิ่งผิดปกติที่เรียกว่า NullPointerException

Null Safety

ซึ่งค่า Null มักจะใช้กับตัวแปรที่ไม่สามารถคาดเดาได้ว่าตัวแปรที่นิยามขึ้นมานั้น มีการเก็บค่าข้อมูลไว้หรือไม่ ?

ถ้าหากไม่มีค่าใดเลย (Null เป็นข้อมูลที่ไม่สามารถระบุค่าได้) จะต้องดำเนินการด้วยวิธีการที่เรียกว่า Null Safety โดยจะมีวิธีการจัดการค่า Null ได้หลายรูปแบบ

Null Safety

Null Safety คือกระบวนการป้องกันการผิดพลาดที่อาจเกิดจากค่าว่าง (null) ซึ่งหมายถึง ไม่มีค่าอะไร ตัวอย่างของค่า null คือ

ไม่มีค่าใดๆ
ไม่เท่ากับค่าว่างใน String
ไม่เท่ากับ 0
ไม่เท่ากับ false
ไม่ตรงกับชนิดข้อมูลใดเลย

จัดการค่า Null

- ตรวจสอบด้วย If Statement หรือ คำสั่ง ?.let{}
- Elvis Operator คือการกำหนดค่าเริ่มต้นให้ตัวแปรนั้นเป็นค่า null ได้ (ใช้เครื่องหมาย ?: ที่ตัวแปร)
- Safe Calls เป็นการเรียกใช้ Property หรือ Method (ฟังก์ชั่น) ที่อาจจะส่งค่า Null กลับมา (ใช้เครื่องหมาย ?)
- Not-Null Assertion Operator กำหนดว่าตัวแปรที่สร้างขึ้นมานั้นไม่สามารถมีค่าเป็น null ได้ (ใช้เครื่องหมาย !!)

Nullable Type

หมายถึง การกำหนดให้ตัวแปรของเรานั้นสามารถเก็บค่า Null ได้ โดยนำเครื่องหมาย ? มาวางต่อท้ายชนิดข้อมูลในตัวแปร ส่งผลให้ตัวแปรของเรานั้นสามารถเก็บค่าปกติอ้างอิงตามชนิดข้อมูลที่กำหนดหรือมีค่า Null ได้

```
var ชื่อตัวแปร : ชนิดข้อมูล? = null;
```

การรับข้อมูล (Input)

คือ คำสั่งสำหรับรับค่าผ่านทางคีย์บอร์ดและเก็บค่าดังกล่าวลงในตัวแปรมีโครงสร้างคำสั่ง ดังนี้

```
var ชื่อตัวแปร  = readLine()
```

*ค่าที่ได้จาก readLine() จะมีชนิดข้อมูลเป็น String ที่เป็นรูปแบบ Nullable

รู้จักกับอาร์เรย์ (Arrays)

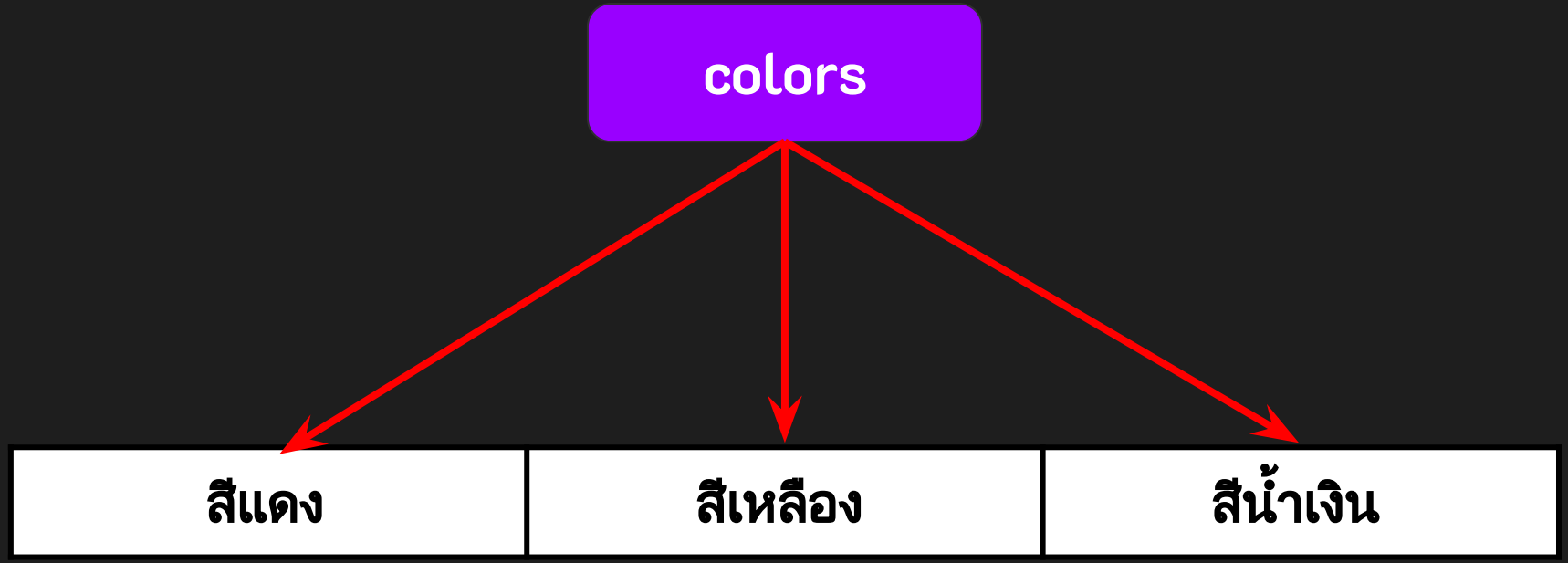
1. ชุดของข้อมูลที่อยู่ในรูปลำดับใช้เก็บค่าข้อมูลให้อยู่ในกลุ่มเดียวกัน

2. ใช้ในการเก็บข้อมูลที่มีลำดับที่ต่อเนื่อง ซึ่งข้อมูลมีค่าได้หลายค่า โดยใช้ชื่ออ้างอิงเพียงชื่อเดียว และใช้หมายเลขกำกับ (index) เพื่อจำแนกความแตกต่างของของการเก็บข้อมูลแต่ละตัว

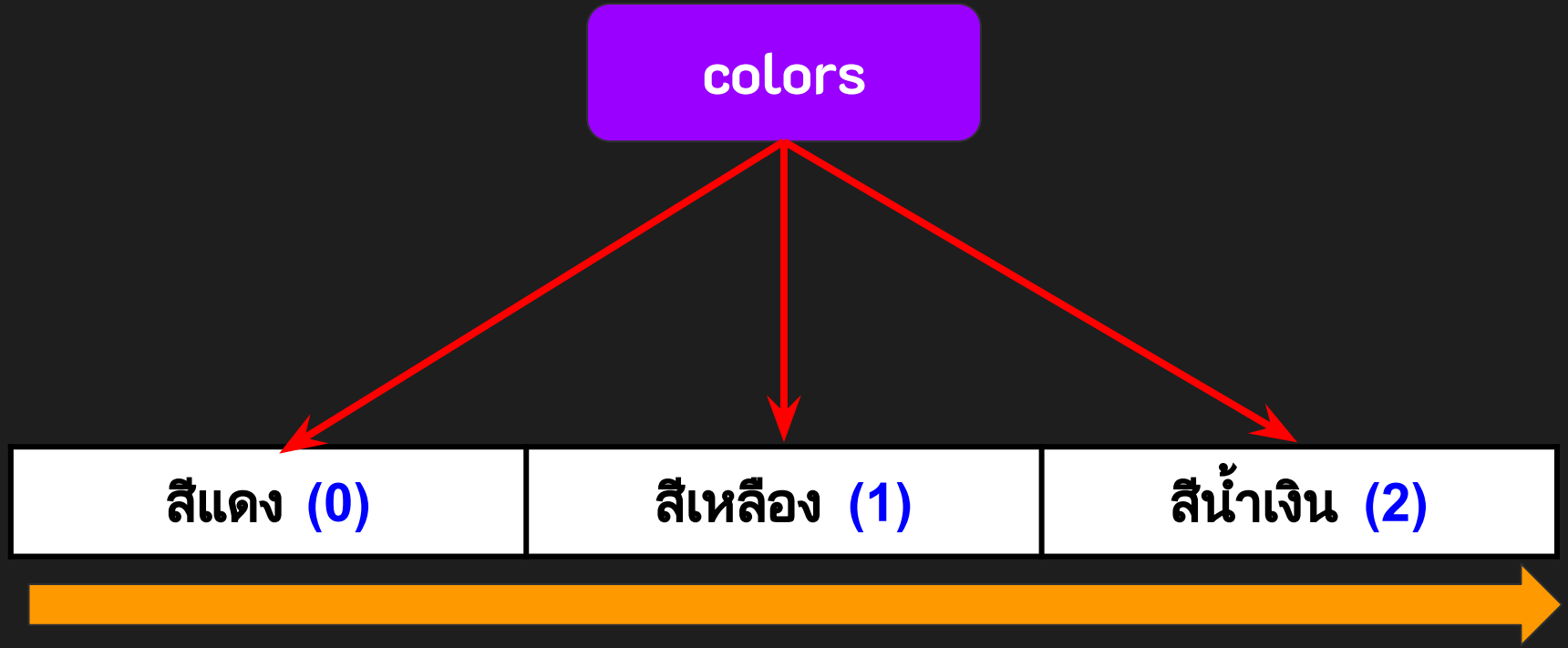
คุณสมบัติของอาร์เรย์ (Arrays)

- ใช้เก็บกลุ่มของข้อมูล
- ข้อมูลที่อยู่ในอาร์เรย์จะเรียกว่าสมาชิก หรือ อิลิเมนต์ (element)
- แต่ละอิลิเมนต์ (element) จะเก็บค่าข้อมูล (value) และ อินเด็กซ์ (Index)
- Index หมายถึงคีย์ของอาร์เรย์ใช้อ้างอิงตำแหน่งของ element เริ่มต้นที่ 0
- สมาชิกที่เก็บในอาร์เรย์สามารถมีชนิดข้อมูลเหมือนกันหรือต่างกันได้
- สมาชิกในอาร์เรย์จะถูกคั่นด้วยเครื่องหมายคอมม่า

การเข้าถึงข้อมูลในอาร์เรย์



การเข้าถึงข้อมูลในอาร์เรย์



สรุปอาร์เรย์

- ใช้เก็บกลุ่มของข้อมูลชนิดเดียวกันหรือต่างกันได้
- ใช้ชื่อเดียวกันในการเก็บข้อมูล
- ใช้หมายเลขกำกับเพื่ออ้างอิงตำแหน่งของข้อมูลในอาร์เรย์
- ข้อมูลที่กำหนดเริ่มต้น สามารถแก้ไขในภายหลังได้
- เหมาะสำหรับเก็บข้อมูลไว้ค้นหาและเรียงลำดับ
- มีขนาดที่แน่นอนไม่สามารถปรับเปลี่ยนขนาดได้

การสร้างอาร์เรย์

- แบบไม่ระบุชนิดข้อมูล (เก็บข้อมูลต่างชนิดกัน)
- แบบระบุชนิดข้อมูล (เก็บข้อมูลชนิดเดียวกัน)

* ใช้ร่วมกับคำสั่ง val เนื่องจากไม่สามารถเปลี่ยนตัวแปรที่อ้างอิงอาร์เรย์ได้ แต่สามารถเปลี่ยนข้อมูลสมาชิกภายในอาร์เรย์ได้

อาร์เรย์แบบไม่ระบุชนิดข้อมูล

โดยการสร้างอาร์เรย์แบบไม่ระบุชนิดข้อมูลนั้น จะใช้งานผ่านฟังก์ชั่น arrayOf() และข้อมูลสมาชิกที่บรรจุอยู่ในอาร์เรย์ดังกล่าวสามารถเก็บข้อมูลชนิดใดก็ได้

อาร์เรย์แบบไม่ระบุชนิดข้อมูล

โครงสร้างคำสั่ง

val ชื่อตัวแปร = arrayOf(ข้อมูลที่ 1 , ข้อมูลที่ 2 , ...)

ตัวอย่าง

```
val data = arrayOf(1 ,“kong”,true)
```

การแก้ไขข้อมูล

โครงสร้างคำสั่ง

อาร์เรย์ที่ต้องการแก้ไขข้อมูล[index] = ค่าข้อมูลใหม่

ตัวอย่าง

```
val data = arrayOf(1 ,“kong”,true)
data[0] = 100
```

การเข้าถึงข้อมูลด้วย For Loop

```
val data = arrayOf(1 ,“kong”,true)
for (item in data){
    println(item)
}
```

อาร์เรย์แบบระบุชนิดข้อมูล

คือการสร้างอาร์เรย์โดยอาศัย คลาส XXXArray หรือ ฟังก์ชั่น XXXArrayOf() สำหรับกำหนดให้การจัดเก็บข้อมูล สมาชิกที่อยู่ในอาร์เรย์นั้นเป็นชนิดเดียวกันทั้งหมด (โดย XXX หมายถึงชื่อชนิดข้อมูล)

สร้างอาร์เรย์ด้วยคลาส (กำหนดขนาด)

IntArray	LongArray	FloatArray
DoubleArray	BooleanArray	CharArray

สร้างอาร์เรย์ด้วยฟังก์ชั่น

intArrayOf()	longArrayOf()	floatArrayOf()
doubleArrayOf()	booleanArrayOf()	charArrayOf()

ฟังก์ชั่นจัดการอาร์เรย์

ชื่อฟังก์ชั่น	คำอธิบาย
count() , size	จำนวนสมาชิกทั้งหมดในอาร์เรย์
contains(element)	ตรวจสอบว่ามีสมาชิกในอาร์เรย์หรือไม่
joinToString (ตัวคั่น)	นำสมาชิกในอาร์เรย์มาเชื่อมต่อกันด้วยตัวคั่นที่กำหนดและแปลงให้อยู่ในรูปแบบสตริง

ฟังก์ชั่นจัดการอาร์เรย์

ชื่อฟังก์ชั่น	คำอธิบาย
indexOf(element)	ตรวจสอบลำดับสมาชิกในอาร์เรย์ (ถ้าเจอให้บอกเลข index , ถ้าไม่เจอจะได้ค่า -1)
last()	สมาชิกตัวสุดท้ายในอาร์เรย์
sortedArray()	เรียงลำดับข้อมูลในอาร์เรย์ (น้อยไปมาก)

ฟังก์ชั่นจัดการอาร์เรย์

ชื่อฟังก์ชั่น	คำอธิบาย
sortedArrayDescending()	เรียงลำดับข้อมูลในอาร์เรย์ (มากไปน้อย)
reversedArray()	เรียงลำดับข้อมูลในอาร์เรย์แบบย้อนกลับ
sum()	คำนวณหาผลรวมสมาชิกทั้งหมดในอาร์เรย์ (กรณีที่เป็นตัวเลข)

ฟังก์ชั่น (Function)

ชุดคำสั่งที่นำมาเขียนรวมกันเป็นกลุ่มเพื่อเรียกใช้งานตามวัตถุประสงค์ที่ต้องการและลดความซ้ำซ้อนของคำสั่งที่ใช้งานบ่อย

ฟังก์ชั่นสามารถนำไปใช้งานได้ทุกที่และแก้ไขได้ในภายหลัง ทำให้โค้ดในโปรแกรมมีระเบียบและใช้งานได้สะดวกมากยิ่งขึ้น

รูปแบบของฟังก์ชั่น (Function)

- **ฟังก์ชั่นมาตรฐาน (Pre-Defined Functions)** คือ ฟังก์ชั่นที่มีอยู่ในภาษา Kotlin สามารถเรียกใช้งานได้ทันที เช่น print() , readLine()

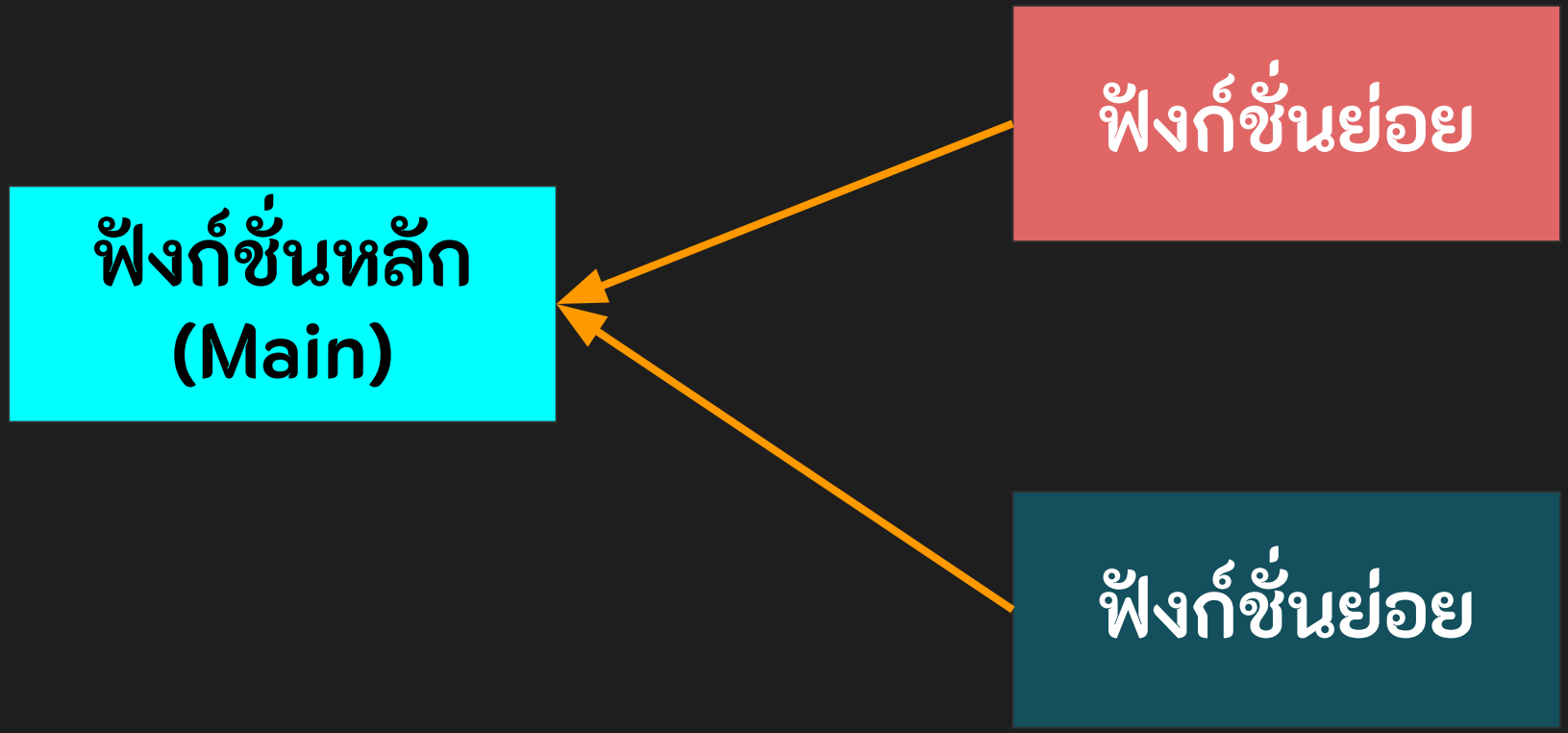
- **ฟังก์ชั่นที่ผู้ใช้สร้างขึ้นมาเอง (User-Defined Function)** คือ ฟังก์ชั่นที่ถูกสร้างขึ้นมาให้ทำงานตามวัตุประสงค์ที่ผู้ใช้ต้องการ

ฟังก์ชั่นหลัก (main)

```
fun main()
{
    //statement
}
```

ฟังก์ชั่น main() คือ ฟังก์ชั่นพิเศษกลุ่มคำสั่งที่อยู่ในฟังก์ชั่นนี้จะถูกสั่งให้ทำงานโดยอัตโนมัติเป็นลำดับแรกเสมอ

ฟังก์ชั่น (Function)



การสร้างฟังก์ชั่น (Function)

- ฟังก์ชั่นแบบปกติ (Unit)
- ฟังก์ชั่นแบบมีพารามิเตอร์ (Parameter)
- ฟังก์ชั่นแบบมีค่าส่งกลับ (Return)
- ฟังก์ชั่นแบบรับและส่งค่า (Parameter & Return)

กฎการตั้งชื่อฟังก์ชั่น

- ชื่อฟังก์ชั่นต้องไม่ซ้ำกัน
- ชื่อฟังก์ชั่นสามารถตั้งเป็นตัวอักษรหรือตัวเลขได้
- ชื่อของฟังก์ชั่นต้องไม่ขึ้นต้นด้วยตัวเลข

ฟังก์ชั่นแบบปกติ (1)

โครงสร้างคำสั่ง

```
fun ชื่อฟังก์ชั่น():Unit{
    // คำสั่งต่างๆ
}
```

การเรียกใช้งานฟังก์ชั่น

```
ชื่อฟังก์ชั่น()
```

ฟังก์ชั่นแบบปกติ (2)

โครงสร้างคำสั่ง

```
fun ชื่อฟังก์ชั่น(){
    // คำสั่งต่างๆ
}
```

การเรียกใช้งานฟังก์ชั่น

```
ชื่อฟังก์ชั่น()
```

ฟังก์ชั่นแบบมีพารามิเตอร์ (Parameter)

โครงสร้างคำสั่ง

```
fun ชื่อฟังก์ชั่น(parameter1 : type ,parameter2 : type,.....){
    // กลุ่มคำสั่งต่างๆ
}
```

การเรียกใช้งานฟังก์ชั่น

```
ชื่อฟังก์ชั่น (argument1,argument2,.....);
```

ฟังก์ชั่นแบบมีพารามิเตอร์ (Parameter)

โครงสร้างคำสั่ง

```
fun ชื่อฟังก์ชั่น(parameter1 : type ,parameter2 : type,.....){
    // กลุ่มคำสั่งต่างๆ
}
```

การเรียกใช้งานฟังก์ชั่น

```
ชื่อฟังก์ชั่น (argument1,argument2,.....);
```

- **อาร์กิวเมนต์** คือ ตัวแปรหรือค่าที่ต้องการส่งมาให้กับฟังก์ชัน (ตัวแปรส่ง)
- **พารามิเตอร์** คือ ตัวแปรที่ฟังก์ชันสร้างไว้สำหรับรับค่าที่จะส่งเข้ามาให้กับฟังก์ชัน (ตัวแปรรับ)

ข้อกำหนดการใช้งานพารามิเตอร์

- ระบุชื่อและชนิดข้อมูลคล้ายกับการประกาศตัวแปร แต่ไม่ต้องระบุคำสั่ง var หรือ val กำกับ
- มีพารามิเตอร์กี่ตัวก็ได้ ขึ้นอยู่กับวัตถุประสงค์ในการใช้งาน โดยจะคั่นพารามิเตอร์แต่ละตัวด้วยเครื่องหมายคอมม่า
- มีชนิดข้อมูลเหมือนกันหรือต่างกันได้
- มีขอบเขตการใช้งานเฉพาะพื้นที่ภายในฟังก์ชั่น (Local Variable)
- สามารถนำไปใช้งานต่างๆในฟังก์ชั่นได้คล้ายกับการใช้งานตัวแปร

ฟังก์ชั่นแบบกำหนดค่าเริ่มต้น

โครงสร้างคำสั่ง

```
fun ชื่อฟังก์ชั่น(parameter : type = ค่าเริ่มต้น){
    // กลุ่มคำสั่งต่างๆ
}
```

ฟังก์ชั่นแบบกำหนดชื่อและลำดับพารามิเตอร์

โครงสร้างคำสั่ง

```
fun ชื่อฟังก์ชั่น(parameter1 : type ,parameter2 : type,.....){
    // กลุ่มคำสั่งต่างๆ
}
```

การเรียกใช้งานฟังก์ชั่น

```
ชื่อฟังก์ชั่น (ชื่อพารามิเตอร์=argument1,ชื่อพารามิเตอร์=argument2,.....);
```

ฟังก์ชั่นแบบมีค่าส่งกลับ

โครงสร้างคำสั่ง

```
fun ชื่อฟังก์ชั่น() : type{
    return ค่าที่จะส่งออกไป (อ้างอิงตามชนิดข้อมูล)
}
```

การเรียกใช้งานฟังก์ชั่น

```
ตัวแปรที่รับค่าจากฟังก์ชั่น = ชื่อฟังก์ชั่น ();
```

ฟังก์ชั่นแบบรับและส่งค่า

โครงสร้างคำสั่ง

```
fun ชื่อฟังก์ชั่น(parameter1,parameter2,.....) : type{
    return ค่าที่จะส่งออกไป (อ้างอิงตามชนิดข้อมูล)
}
```

การเรียกใช้งานฟังก์ชั่น

```
ตัวแปรที่รับค่าจากฟังก์ชั่น = ชื่อฟังก์ชั่น(argument1,argument2..);
```

Function Overloading

ฟังก์ชั่นแบบโอเวอร์โหลด หมายถึง การสร้างฟังก์ชันที่มีชื่อเหมือนกันแต่สามารถรับพารามิเตอร์จำนวนต่างกันพร้อมคืนค่าที่แตกต่างกันได้

กฎการตั้งชื่อฟังก์ชั่น

- ชื่อฟังก์ชั่นต้องไม่ซ้ำกัน
- ชื่อฟังก์ชั่นสามารถตั้งเป็นตัวอักษรหรือตัวเลขได้
- ชื่อของฟังก์ชั่นต้องไม่ขึ้นต้นด้วยตัวเลข

ข้อกำหนด

- ชื่อฟังก์ชั่นต้องเขียนเหมือนกัน
- ถ้าชนิดข้อมูลของพารามิเตอร์เหมือนกัน จำนวนพารามิเตอร์ต้องไม่เท่ากัน
- ถ้าชนิดข้อมูลของพารามิเตอร์ต่างกัน จำนวนพารามิเตอร์เท่ากันหรือไม่เท่ากันก็ได้
- ชนิดข้อมูลส่งกลับสามารถเหมือนกันหรือต่างกันได้

ตัวอย่างที่ 1

1. fun showData() : String
2. fun showData(value:String) : String
3. fun showData(value:Int) : Int

ตัวอย่างที่ 1

1. fun showData() : String
2. fun showData(value:String) : String
3. fun showData(value:Int) : Int

ตัวอย่างที่ 2

1. fun total (a:Int,b:Int) :Int
2. fun total (a:String,b:String) : Double
3. fun total (a:Double,b:Double,c:Double) : Double

ตัวอย่างที่ 2

1. fun total (a:Int,b:Int) :Int
2. fun total (a:String,b:String) : Double
3. fun total (a:Double,b:Double,c:Double) : Double

Variable Arguments

- การสร้างพารามิเตอร์ที่ทำงานในฟังก์ชั่นแบบไม่จำกัดจำนวน ใช้ในกรณีที่ไม่สามารถคาดเดาได้ว่าจะมีการส่งอากิวเมนต์เข้ามาใช้งานในฟังก์ชั่นกี่จำนวน

- การสร้างพารามิเตอร์ดังกล่าวจะใช้ร่วมกับคำสั่ง vararg นำหน้าชื่อพารามิเตอร์ เช่น vararg data:Int

Lambda Expression

หมายถึง การสร้างฟังก์ชั่นแบบไม่มีชื่อ

โครงสร้างคำสั่ง

val ชื่อตัวแปร : (ชนิดข้อมูลพารามิเตอร์) ->

ชนิดข้อมูลค่าส่งกลับ = {พารามิเตอร์ -> คำสั่งต่างๆ}

Lambda Expression

โครงสร้างคำสั่ง

val ชื่อตัวแปร : (ชนิดข้อมูลพารามิเตอร์) -> ชนิดข้อมูลค่าส่งกลับ = {พารามิเตอร์ -> คำสั่งต่างๆ}

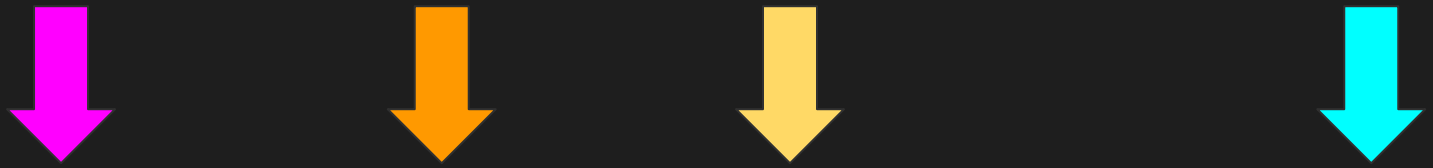
ตัวอย่าง

```
val total : (Int,Int) -> Int = {a:Int , b:Int -> a+b}
```

Lambda Expression

โครงสร้างคำสั่ง

val ชื่อตัวแปร : (ชนิดข้อมูลพารามิเตอร์) -> ชนิดข้อมูลค่าส่งกลับ = {พารามิเตอร์ -> คำสั่งต่างๆ}



```
val total : (Int,Int) -> Int = {a:Int , b:Int -> a+b}
```

Lambda Expression

โครงสร้างคำสั่ง

```
val total : (Int,Int) -> Int = {a:Int , b:Int -> a+b}
```

ตัวแปร total มีชนิดข้อมูลเป็น (Int,Int)->Int

จะเรียกตัวแปร total ว่าเป็นข้อมูลประเภทฟังก์ชั่น (Function Type)

It Keyword

กรณีที่ฟังก์ชั่นเขียนแบบ Lambda และมีพารามิเตอร์แค่ตัวเดียว เราไม่จำเป็นต้องเขียนชื่อพารามิเตอร์และเครื่องหมาย -> กำกับ สามารถใช้คำสั่ง it เป็นตัวแทนของพารามิเตอร์ได้เลย

Higher Order Function

หมายถึง ฟังก์ชั่นที่มีพารามิเตอร์ในลักษณะของ Function Type (Lambda)

ขอบเขตตัวแปร

- Local variable ตัวแปรที่ประกาศอยู่ภายในพื้นที่ฟังก์ชั่นมีขอบเขตการทำงานตั้งแต่จุดเริ่มต้นไปจนถึงจุดสิ้นสุดของฟังก์ชั่นจะถือได้ว่าฟังก์ชั่นนั้นเป็นเจ้าของตัวแปรนั้น ฟังก์ชั่นอื่นจะไม่สามารถเรียกใช้งานตัวแปรนี้ได้

ขอบเขตตัวแปร

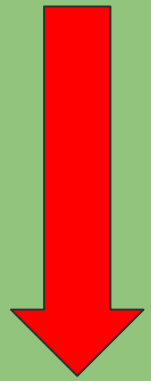
- Global variable ตัวแปรที่ประกาศอยู่นอกฟังก์ชั่นมีขอบเขตการทำงานตั้งแต่จุดเริ่มต้นไปจนถึงจุดสิ้นสุดของไฟล์ที่ประกาศใช้ นั่นหมายถึงตัวแปรดังกล่าวนั้นเป็นสาธารณะ ไม่มีฟังก์ชั่นใดเป็นเจ้าของ ทุกฟังก์ชั่นสามารถเรียกใช้งานตัวแปรนี้ได้

Local Variable

```
fun main(){

   var balance = 10000




}
```

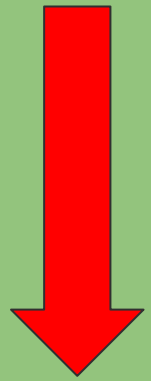
Local Variable

```
fun main(){

    var balance = 10000

}
```



Local Variable

```
fun main(){

    var balance = 10000

}
```

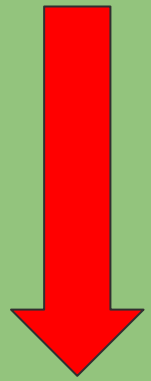


```
fun deposite (amount:Int){

        var value = amount
}
```

```
fun withdraw (amount:Int){

        var value = amount
}
```

Local Variable

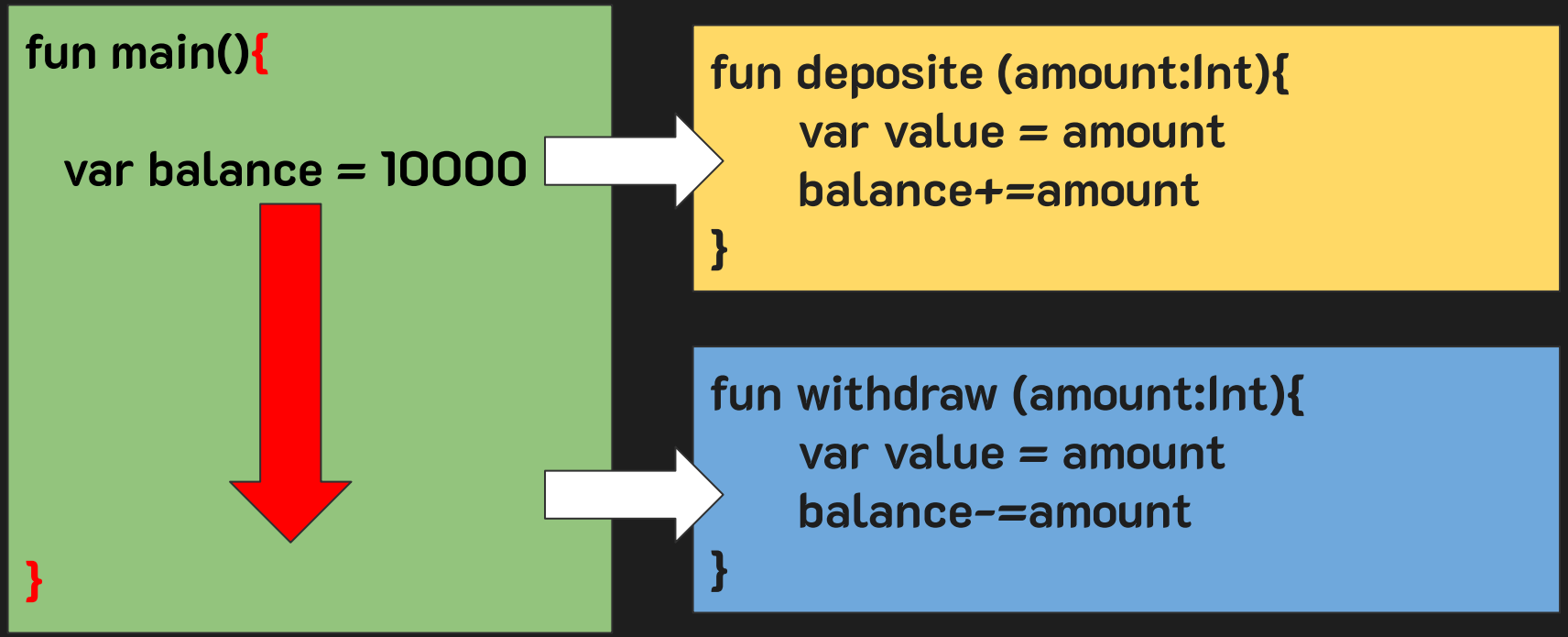
```
fun main(){

    var balance = 10000;

}
```



```
fun deposite (amount:Int){

        var value = amount
}
```

```
fun withdraw (amount:Int){

        var value = amount
}
```

Local Variable



Local Variable

```
fun main(){

    var balance = 10000

}
```

```
fun deposite (amount:Int){
        var value = amount
        balance+=amount
}
```

```
fun withdraw (amount:Int){
        var value = amount
        balance-=amount
}
```

Global Variable

```
var balance = 1000
fun main(){
 .......
}

fun deposite(amount:Int){
 .......
}
fun withdraw(amount:Int){
 .......
}
```

Global Variable

```
var balance = 1000 //global variable
fun main(){
    println(balance)
}

fun deposite(amount:Int){
    balance+=amount
}
fun withdraw(amount:Int){
    balance-=amount
}
```

Exception

การที่โปรแกรมทำงานบางอย่างแต่เกิดข้อผิดพลาดขึ้น แล้วโปรแกรมไม่สามารถจัดการข้อผิดพลาดนั้นได้ ซึ่งทำให้เกิดสิ่งผิดปกติหรือ Exception ส่งผลทำให้โปรแกรมหยุดทำงาน

โดยในภาษา Kotlin สามารถจัดการสิ่งผิดปกติได้โดยใช้คำสั่ง Try..Catch หรือ Throw เหมือนในภาษา Java เลย

Try...Catch

```
try{
    // ลองทำคำสั่งในนี้
}catch(e: Exception){
    // ถ้าเกิดข้อผิดพลาดจะมาทำตรงส่วนนี้
}
```

การแสดงรายละเอียดข้อผิดพลาด

Property / Function	ความหมาย
message	แสดงข้อความบอกสาเหตุข้อผิดพลาด
toString()	แสดงข้อความบอกสาเหตุข้อผิดพลาด พร้อมแพ็คเกจของคลาสที่เกี่ยวข้อง
printStackTrace()	แสดงรายละเอียดข้อผิดพลาด ที่เกี่ยวข้องทั้งหมด

Finally

เมื่อเกิดข้อผิดพลาด หรือไม่เกิดก็จะทำงาน คำสั่งในส่วนนี้ทุกครั้งคำสั่งที่ระบุมักจะเป็นคำสั่งที่ทำงานส่วนที่สำคัญของโปรแกรม

```
try{
    // ลองทำคำสั่งในนี้
}catch(e: Exception){
    // ถ้าเกิดข้อผิดพลาดจะมาทำตรงส่วนนี้
} finally {
    // คำสั่งต่างๆ
}
```

Throw

เมื่อเกิดข้อผิดพลาด (Exception) สามารถจัดการโดยการส่งต่อการเกิดข้อผิดพลาดดังกล่าวไปให้ส่วนอื่นจัดการแทนได้ โดยใช้คำสั่ง throw ในตำแหน่งที่อาจจะเกิด Exception