Table of Contents

動機

改善開發速度及穩定度

開發速度

開發速度與穩定度是相對互斥的兩個方向，在 Java 中提高穩定度可以使用框架或者決定團隊的 coding style 來處理這件事

相對的開發者就必須要符合框架規則或者熟悉團隊規定的 coding style，這樣的做法很可能就會造成開發速度降低

程式穩定性

追求開發速度，很多事情就只能睜一隻眼閉一隻眼的放過，反正功能先出來，優化之後做

但是需求會一直來，根本不會有時間做優化，導致債築高牆，堆到某個程度也沒人可以改

最後就成為補丁般的程式，各種問題只能人工上線針對個案處理

預備知識

具備以下知識有助於更滑順的理解此份報告內容

- 任何一款靜態語言
- Java 8+
- TypeScript

選擇條件

Koltin 的程式結構比 Java 簡潔，但是語言特性上比卻比 Java 嚴謹，想較於其他語言而言，對 incrte 來說可以算是有機會同時提升開發速度與穩定性的語言

- Kotlin 可以幫助開發者在撰寫程式碼時，比較容易注意到可能發生 NullPointer 的地方，但是又不需要一直在那邊 if null

  kotlin

```
var s:String? = null
println("string length:${s?.length}")
```

java

```
String s = null;
if (null != s) {
  System.out.println("string length:"+s.length());
} else {
  System.out.println("string length:null");
}
```

- 支援 High Order Function : 可以撰寫更靈活的程式碼，在 Java 上要實現需要做很多工

  kotlin

```
var intOpt = fun(x:Int, y:Int, f:(Int,Int)->Int):Int = f(x,y
var optAdd = fun(x:Int, y:Int):Int = x + y
val result = intOpt(1,2, optAdd)
println(result)
```

  java

```
public class MainJava {
    public static Integer intOpt(Integer x, Integer y, BiFun
        return f.apply(x, y);
    }
    public static void main(String[] args) {
        BiFunction<Integer, Integer, Integer> f = (x,y)->x+y
        Integer result = intOpt(1,2, f);
        System.out.println(result);

    }
}
```

  > java: 微臣辦不到啊。。。

- 兼容 Java : 編譯後產生 class 檔一樣跑在 JVM 上，程式上也能和 Java 互相呼叫，對目前 incrte 技術親合度很高

  kotlin (承接上例)

```
val result = MainJava.intOpt(1,2, { x, y -> x+y })
println(result)
```

原始碼檔案

以 `.kt` 為結尾的檔案，經過 `kotlinc` 編譯後產生 `.class` 檔

語法

- 預設不需要 `;` 做結尾，除非將多行程式寫在同一行
- `main` 方法為程式進入點，預設為含參數方法 `main(args:ArrayList<String>)` ，如果不需要處理指令參數則也可以省略參數宣告，如 `main()`
- Kotlinc 編譯原則是將檔名當成一個 class，所有在原始碼中宣告的類別都是此類別的內部類別，預設 class 名稱是 `檔名+Kt` ，如 `HelloWorld.kt` -> `HelloWorldKt.class`

變數宣告

- 格式： `var/val 變數名稱:型別`
- `var` ：表示此變數為可異動變數
- `val` ：表示此變數為不可異動變數 (常數)

```
val DEPARTMENT = "CSO"
DEPARTMENT = "PDT" // <-- error
var version:String = ""
version = "v3"
```

型別

所有型別都是參考型別

> 沒有 java 中的 primitive type

整數

`Int` 、 `Long` 、 `Short` 長度分別為 32 、64、16 位元

> 長整數的宣告只允許使用大寫 `L`

位元組

`Byte` 長度為 8 位元

布林值

`Boolean` 長度為 1 位元

浮點數

`Float` 、 `Double` 長度分別為 32、64 位元

字元

`Char` 長度為 16 位元

字串

`String`

字串模板

在 `""` 之間如果需要呼叫變數，可以使用 `$變數名稱` 或者 `${變數.屬性名稱}` 來將成員嵌入字串中

> 不需要像 java 有那麼多字串串接的貓毛

如果要串接錢字號 `$` ，則需要使用字面值宣告 `${'$'}` 來使用

多行文字

在三個雙引號之間的文字會被視為多行文字

> 不需要像 java 一樣 + 來 + 去

```
"""
第一行
第二行
第三行
""";
```

空值

所有變數預設不可為空**!!!**

空值宣告

如果某變數確實可能為空，那必須將此變數宣告為**可空狀態**

可空狀態

在變數宣告的型別後方加上問號 `?` 來表示此變數可能為空值

編譯器對可不可空變數的處理不同，編譯器對於不同狀態的變數有不同的處理方式

在編譯期就可以比較安全的撰寫程式

同一個型別可不可空是兩種不同的狀態，即：`String?` 與 `String` 是不同的

安全操作

阻擋呼叫

呼叫可空狀態的變數時，如果該變數真的是空值，則不會繼續往下呼叫，避免噴出 NullPointerException

```
var nullString:String?
println(nullString?.length) // <-- 不會真的去呼叫 length : 不會噴錯
```

阻擋賦值

可空變數不可賦值給非空變數，編譯器會直接報錯，就算該可空變數不為空

```
var nullString:String? = "123"
var s:String = nullString // <-- error
var l:Int = nullString?.length // <-- error
```

強制非空

要將可空變數強制當作非空存取，可以使用 `!!` 符號

> 設計上盡量避免這樣使用

```
var nullString:String? = "123"
var s:String = nullString!! // <-- it's ok, but not recommand
```

最大父類別

`Any`

類似 java 中的 `Object` 類別，但是少了一些方法

是所有類別的父類別

最小子類別

`Nothing`

和 `Any` 相反，是所有類別的子類別

> 就算你不想要，他一定是你的子類別

"無" 型別

`Unit`

就是 void

型別轉換

直接賦值不會自動轉換

```
var b:Byte = 1
var i:Int = b // <-- error
```

運算會自動想上轉型

```
var i:Int = 100
var d:Double = 20.0
val result:Double = i - d // it's ok
```

強制轉型

使用 `as` 可以強制轉型

```
var b:Byte = 1
var i:Int = b as Int
```

安全轉型

使用 `as?` 配上可空宣告來達成安全轉換，如果轉換錯誤或者無法轉換該變數就為空

```
var number:Int? = "foo" as? Int // number 是空，但不會噴錯
```

判定後自動轉換

使用 `is` 判定後的區塊內該變數會自動轉換成判定的型別

```
var foo:Any = "I am a string"

if (foo is String) {
  println(foo.toUpperCase()) // <-- 區塊內 foo 就當作是 String 型
}
```

表達式

在 kotlin 中只要看到某些操作子可以做表達式使用

表示該操作子可以寫在賦值操作符 `=` 的右方

最後會回傳該表達式的最後一行執行結果，且不需要 `return` 關鍵字

例如：

```
val s = if (true) "Hello" else "Kotlin" // s = "Hello"
val s2 = if(false) {
  "Hello"
} else {
  "Kotlin"
} // s2 = "Kotlin"
```

一元操作符

正負操作子

符號	等價方法
`+`	`unaryPlus()`
`-`	`unaryMinus()`

```
var a = 10
var b = -a // b = -10
var c = a.unaryMinus() // c = -10
```

遞增抵減操作子

符號	等價方法
`++`	`inc()`
`--`	`dec()`

否定操作子

符號	等價方法
`!`	`not()`

二元操作子

四則運算

符號	等價方法
`+`	`plus()`
`-`	`minus()`
`*`	`times()`
`/`	`div()`
`%`	`rem()`

複合操作子

符號	等價方法
`+=`	`plusAssign()`
`-=`	`minusAssign()`
`*=`	`timesAssign()`
`/=`	`divAssign()`
`%=`	`remAssign()`

比較操作子

符號	等價方法
`>`	`compareTo()`
`<`	`compareTo()`
`>=`	`compareTo()`
`<=`	`compareTo()`

> 等價方法：`compareTo()`
>
> - 左大於右：返回大於 0
> - 右大於左：返回小於 0
> - 左右相等：返回等於 0

位元運算子

> kotlin 中沒有位元運算子的符號，只提供等價方法

符號	等價方法	說明
`&`	`and()`	交集
`|`	`or()`	聯集
`!`	`inv()`	反向
`^`	`xor()`	邏輯互斥或
`<`	`shl()`	左移位元
`>`	`shr()`	右移位元
`>>`	`ushr()`	保持最左方位元(正負號)不變，其餘位元右移

包含操作子

關鍵字：`in`

判斷左方目標是否包含於右方目標， `a in b` 類似 java 中的 `b.contains(a)`

不包含符號： `!in`

內容相等(等同)

關鍵字： `==`

相當於 java 中的 `equals` 方法，比較兩個目標內容是否相同

不等同符號： `!=`

參考相等(等於)

關鍵字： `===`

相當於 java 中的 `==` 運算，比較兩個目標是否指向同一個參考對象

不等於符號： `!==`

貓王操作子

關鍵字： `?:`

當第一個目標為空值時，返回第二個目標

語法 : 第一目標 `?:` 第二目標

可以做連鎖空值表達式

```
var a:String = null ?: null ?: null ?: null ?: "hi I'm a"
```

結果 `a = "hi I'm a"`

索引操作子

符號 `[]` 可以對集合型別存取部分數據

取值： `a[index]` -> 相當於 getter 方法 賦值： `a[index] = x` -> 相當於 setter 方法

區間操作子

符號 `..` 可以創建區間物件，相當於第一個目標呼叫 `rangeTo()` 方法

```
var range = 1..3
var range2 = 1.rangTo(3)
```

覆寫預設操作子

kotlin 允許覆寫預設操作子的行為，當然也能為自定義的類別撰寫特殊的操作子行為

例如使用 `+` 符號實現 黃色 + 藍色 = 綠色 的範例

```
class Blue {
    operator fun plus(y: Yellow):Green = Green()
}
class Yellow {
    operator fun plus(b:Blue):Blue = Blue()
}
class Green

fun main() {
    val blue:Blue = Blue()
    val yellow:Yellow = Yellow()
    val green = blue + yellow
}
```

條件

if 條件

語法：

```
if (條件) {
} else if(條件) {
} else {
}
```

when 條件

when 宣告 (when-statement)

類似 java 的 switch-case 宣告

語法：

```
when(變數) {
    條件一 -> 變數滿足條件一執行區塊
    條件二 -> 變數滿足條件二執行區塊
    else -> 其他條件執行區塊
}
```

當使用 **when** 宣告的時候，`else` 區塊不可省略

when 區塊 (when-block)

可以當作是加強的 if-else 語法

語法：

```
when {
    條件一 -> 變數滿足條件一執行區塊
    條件二 -> 變數滿足條件二執行區塊
}
```

當使用 **when** 區塊的時候，`else` 區塊非必要

這種特性使得 when 區塊可以處理多個完全不相關的條件

例如

```
when {
    x + y == 5 -> println("x+y=5")
    age < 10 -> println("age under ten")
    else -> println("none of above") // 非必要
}
```

when 使用展示

when 的條件可以有很多種宣告方式，比 switch-case 還要強很多

基礎

```
when(lang) {
    "Java" -> "Java picked"
    "Kotlin" -> "Kotlin rocks"
    "Scala" -> "It's Scala"
    else -> "$lang selected"
}
```

多值共結果

```
when(lang) {
    "Java", "Kotlin" -> "JVM base"
    "js", "ts" -> "V8 base"
    else -> "what is $lang"
}
```

範圍

```
when(score) {
    in 90..100 -> "A"
    in 80..89 -> "B+"
    in 60..79 -> "B"
    !in 0..100 -> "score incorrect"
    else -> "F"
}
```

型別

```
when(變數) {
    is Boolean -> "布林值"
    is Int -> "整數"
    is String -> "字串"
    else -> "其他型別"
}
```

綜合應用

```
fun testWhen(target:Any):String {
  return when(target) {
    0 -> "object equals match"
    3, 10 -> "or match"
    in 11..20 -> "range match"
    is Date -> "type match"
    !in 4..30 -> "range not match"
    else -> "default"
  }
}
testWhen(0) // "object equals match"
testWhen(3) // "or match"
testWhen(11) // "range match"
testWhen(33) // "range not match"
testWhen(Date()) // "type match"
testWhen("whatever") // "default"
```

迴圈

kotlin 中迴圈宣告主要 `for` 和 `while`，和條件不同的是迴圈宣告不可當作表達式

while

分為 while 與 do-while，用法和 java 相同

語法：

```
while(條件) {
}

do {
} while(條件)
```

for

kotlin 中的 for 很特別，沒有 `for(初始;條件;跌代)` 這種用法

注意是 沒有!!

語法 :

```
for (區域變數 in <Iterable>對象) {
}
```

repeat

repeat 區塊單純是實現重複執行 n 次區塊內的程式，雖然同樣的功能也能用 for 或 while 實現，不過這種簡單的流程用 repeat 看起來更簡潔

語法 :

```
repeat(次數) {
}
```

迴圈標籤

可以在迴圈之前做標籤，給內部的 `continue` 或 `break` 使用

```
outer@
for(i in 1..3) {
    for(j in 11..13) {
        if (j % 2 == 0) {
            continue@outer
        }
    }
}
/**
i=1, j=11
i=2, j=11
i=3, j=11
*/
```

陣列 (Array)

kotlin 中陣列分為兩種，一種是**類型元素陣列**，這種陣列內的元素都是參考類型的物件，另一種是 **基本類型元素陣列**

因為在 kotlin 中已經把 java 的基礎類型全部用包覆類別包起來，如果需要創建**基礎類型元素的陣列**，需要使用另外一種宣告方式。

1. 類型元素陣列

- 元素可空： `arrayOfNulls<類型>(長度)` `arrayOfNulls<Int>(3)` -> `[0,0,0]`
- 宣告並初始化陣列： `arrayOf<類型>(元素)` `arrayOf<Int>(1,2,3)` -> `[1,2,3]`
- 陣列建構式： `Array<類型>(長度) { it(元素索引) }` `Array<Int>(3) { it + 1 }` -> `[1,2,3]`
- 複合類型元素： `arrayOf(元素)` `arrayOf(1, "hello", true)` : 陣列元素類型會以 `Any` 來置入

2. 基本類型元素陣列

基本類型陣列宣告有各自對應的創建方法：

- IntArray
- LongArray
- FloatArray
- DoubleArray
- BooleanArray
- ShortArray
- ByteArray
- CharArray

創建方式

- 指初始化長度： `陣列方法(長度)` `IntArray(3)` -> `[0,0,0]`
- 陣列建構式： `陣列方法(長度){ it(元素索引) }` `IntArray(3) { it }` -> `[0,1,2]`
- 宣告並初始化陣列：kotlin 內建基本類型陣列方便創建的方法
  - `intArrayOf(元素)` -> `intArrayOf(1,2,3)` -> `[1,2,3]`
  - `charArrayOf(元素)` -> `charArrayOf('h','e','l','l','o')` -> `['h','e','l','l','o']`

- 其他類型以此類推：
  `longArrayOf` 、 `shortArrayOf` 、 `byteArrayOf` 、 `doubleArrayOf` 、 `floatArrayOf` 、 `booleanArrayOf`

內建屬性

- 可用於 `for-in` 宣告來輪巡元素
- 內建 `indices` 屬性可以拿到輪巡索引值
- 內建 `withIndex()` 方法可以輪巡元素(value)+索引值(index)的數組(tuple)

  範例

```
val array = intArrayOf(1,2,3)
for (tuple in array.withIndex()) {
  println("array[${tuple.index}] = ${tuple.value}")
}
```

列表 (List)

列表和陣列最大的差別是列表長度是可變動的，而陣列不行。

kotlin 中的列表分為唯讀和可變兩種，唯讀列表在創建後就只能被讀取，不能做任何調整。

如果在創建的時候給定了元素內容，則不需要宣告元素類型，kotlin 會自動配適。

1. 唯讀列表

- 初始化元素： `listOf<類型>(元素)` `listOf("Java", "Groovy", "Kotlin")` -> `["Java","Groovy","Kotlin"]`
- 自動排除空元素： `listOfNotNull<類型>(元素)` `listOfNotNull("Java", null, "Kotlin")` -> `["Java", "Kotlin"]`
- 列表建構式： `List<類型>(長度) { it(元素索引) }` `List(3) { 'a' + it }` -> `['a','b','c']`

2. 可變列表

- 初始化元素： `mutableListOf<類型>(元素)`
- 列表建構式： `MutableList<類型>(長度) { it(元素索引) }`

底層基於 **java ArrayList** 的列表

kotlin 中的 MutableList 底層的實現可以依照 kotlin 版本不同而不同，最早的版本 MutableList 的底層是用 java 的 LinkedList 實現，後來才改成 ArrayList，如果不希望因為 kotlin 版本問題而需要明確創建 java 的 ArrayList 結構，kotlin 也有提供明確的宣告方法

宣告方法：`arrayListOf<類型>(元素)`

空列表

空列表是不包含任何元素的唯讀列表，通常用於預設回傳值，回傳時不需要再指定類型

宣告式：`emptyList<類型>()`

```
fun getList():List<Int> {
  if (somecondition) {
    return listOf(1,2,3)
  }
  return emptyList() // <- 無須再指定類型
}
```

內建屬性

- `size`：取得列表長度
- `get(索引)`：取得索引元素，也可以使用索引存取子 `[索引]` 的方式
- `add(元素)`：添加元素
- `set(索引, 元素)`：覆蓋列表中索引所在元素，也可以使用索引存取子 `[索引] = 元素` 的方式
- `remove(元素)`：刪除列表中指定元素，如果元素不存在則對列表沒有任何影響
- `removeAt(索引)`：刪除列表中指定索引的元素，如果索引超過列表範圍則會拋出 IndexOutOfBoundsException
- 可用於 `for-in` 宣告來輪巡元素
- 內建 `indices` 屬性可以拿到輪巡索引值
- 內建 `withIndex()` 方法可以輪巡元素(value)+索引值(index)的數組(tuple)

集合 **(Set)**

- 元素不可重複
- 分為唯讀和可變集合
- 唯讀集合宣告：`setOf<類型>(元素)`，同列表以此類推
- 可變集合宣告：`mutableSetOf<類型>(元素)`，同列表以此類推
- 底層基於 java HashSet 的集合：`hashSetOf<類型>(元素)`，kotlin 底層預設以 LinkedHashSet 實現
- 底層基於 java LinkedHashSet 的集合：`linkedHashSet<類型>(元素)`，道理同上，依照 kotlin 版本不同，預設的 `setOf` 或者 `mutableSetOf` 底層實現可能不同，如果需要特別指定，可以這樣使用
- 底層基於 java treeSet 的集合：`sortedSetOf<類型>(元素)`

空集合

道理同空列表

宣告式：`emptySet<類型>()`

內建屬性

- `size`：取得集合大小
- `add(元素)`：添加元素
- `remove(元素)`：刪除元素
- 可用於 `for-in` 宣告來輪巡元素

區間 (Range)

- 區間中的元素是連續的
- 區間只有儲存起始值和結束值
- 只有：IntRange、LongRange、CharRange 三種區間類型
- 自動配適創建：`開始值.rangeTo(結束值)` `'a'.rangeTo('c')` -> `['a', 'c']`
  - 不包含結束值：`開始值.until(結束值)` `'a'.until('c')` -> `['a','c')`
  - 反向宣告：`開始值.downTo(結束值)` `3.downTo(1)` -> `[3,2,1]`
- 區間操作符號：`開始值..結束值` `'a'..'c'` -> `['a', 'c']`

內建屬性

- `step(間隔)`：區間元素間間隔預設為 1，這個方法可以改變元素間間隔
- `reversed()`：開始值和結束值反向

- `contains(元素)` ：判斷該元素是否包含於區間中

序列 (Sequence)

序列是一組元素間有某種關係的數值數據

- 基礎宣告： `sequenceOf(元素)` `sequence(1,2,3,5)` -> `1,2,3,5`
- 建構式： `generateSequcene(種子){ it(種子或上一個元素) }` 範例： `[3,8,13,18]`

```
val seq = generateSequence(3) {
  val t = it + 5 //第一個 it 會是種子，之後就會是上一個回傳值
  if (t > 20) { // 當超過 20 回傳 null，sequence 就會終止
    null
  } else {
    t
  }
}
```

映射 (Map)

同上，kotlin 中的 mapping 也分為唯讀與可變兩種類型，預設情況，建立出來的會是 java.util.LinkedHashMap 的物件 (原理同 Set)

1. 唯讀映射

- 初始映射： `mapOf<key類型, value類型>(元素)`
  - 使用 Pair 類型初始： `Pair(key, value)`

```
val map = mapOf(Pair("name", "Tom"), Pair("national",
```

  - 使用 `to()` 方法初始： `key to value`

```
val map = mapOf("name" to "Tom", "national" to "USA")
```

2. 可變映射

- 初始映射： `mutableMapOf<key類型, value類型>(元素)`
  - 初始方法和唯讀相同

底層基於 **java.util.HashMap** 映射

宣告方法：`hashMapOf<key類型, value類型>(元素)`

底層基於 **java.util.LinkedHashMap** 映射

宣告方法：`linkedHashMapOf<key類型, value類型>(元素)`

底層基於 **java.util.TreeMap** 映射

宣告方法：`sortedMapOf<key類型, value類型>(元素)`

空映射

宣告方法：`emptyMap<key類型, value類型>()`

和所有空集合的空實例相同，主要用於回傳時不需要再指定類型

內建屬性

- `size`：取得映射元素個數
- `get(key)`：取得 key 對應的 value 值，也可以使用索引存取子 `[key]` 來取得
  - 如果 key 不存在，則會返回空值
  - `getValue(key)`：key 不存在會拋出例外而不是返回空值
  - `getOrDefault(key, default value)`：key 存在回傳 value 值，不存在回傳預設
- `put(key, value)`：添加或修改 key 對應的 value 值，也可以使用索引操作子 `[key] = value` 來執行
  - `putIfAbsent(key, value)`：如果 key 不存在，則會添加；如果已經存在，則不會修改
- `remove(key)`：刪除 key 與對應的 value
- 可用於 `for-in` 宣告來輪巡元素
  - Entry 輪巡：`for (entry in map) { println("${entry.key}:${entry.value}") }`
  - 解構語法輪巡：`for ((k, v) in map) { println("$k:$v") }`

集合類型關係圖

看不到圖請點

```
classDiagram
class Collection{
  Int size
  isEmpty() Boolean
  contains(element) Boolean
  containsAll(elements) Boolean
  iterator() Iterator
}
<<interface>> Collection
class List
<<interface>> List
class MutableList
<<interface>> MutableList
class ArrayList
class Set
<<interface>> Set
class MutableSet
<<interface>> MutableSet
class HashSet
class LinkedHashSet
class MutableCollection{
  iterator() MutableIterator
  add(element) Boolean
  addAll(elements) Boolean
  remove(element) Boolean
  removeAll(elements) Boolean
  retainAll(elements) Boolean
  clear() Unit
}
<<interface>> MutableCollection
Collection <|-- List
List <|-- MutableList
MutableList <|.. ArrayList
Collection <|-- Set
Set <|-- MutableSet
MutableSet <|.. HashSet
HashSet <|-- LinkedHashSet
Collection <|-- MutableCollection
MutableCollection <|-- MutableList
MutableCollection <|-- MutableSet
```

集合類型表

類別	可變	元素可重複	索引存取	動態載入	底層實現
List<>	N	Y	Y	N	java.util.Arrays$ArrayLis
MutableList<>	Y	Y	Y	N	java.util.ArrayList
Set<>	N	N	N	N	java.util.LinkedHashSet
MutableSet<>	Y	N	N	N	java.util.LinkedhashSet
Range	N	Y	N	Y	Kotlin 原生類型
Sequence	N	Y	N	Y	Kotlin 原生類型

一般方法

宣告格式：

```
[存取範圍] fun [方法名稱](參數列表) [:回傳值類型] {
}
```

無返回值方法可以宣告回傳值類型為 `Unit` 或者就直接不需要宣告回傳值類型

多個回傳值

kotlin 預設有 Pair 與 Triple 類型，可以讓方法回傳兩個或三個回傳值

> 在 2012 年的時候，Kotlin 是有 Tuple 類型的，但在後面的版本移除了 官方解釋是有了 Pair 和 Triple 類型已經足夠了，如果回傳值數大於三個 那應該要建立一個類型當回傳類型

參數

> 在中文翻譯上，我們並不常去區別宣告和呼叫時的參數名詞上的差異，但其實這兩個是不同的 在宣告時的參數，大陸翻譯叫 "形式參數" 或 "型參"，台灣這邊就叫做 "參數"，英文是 parameter 在實際呼叫時帶入的值，大陸翻譯叫做 "實際參數" 或 "實參"，台灣這邊正式名稱叫做 "引數"，英文是 argument 例如
>
> ```
> fun Foo(i:Int, f:Float) {}
> Foo(1, 2.0)
> ```
>
> 其中 `i` 、 `f` 是 parameter， `1` 和 `2.0` 是 argument

命名參數

傳統方法呼叫參數對應是用 **位置對應** 來執行，意思是

```
fun greeting(name:String, word:String) {
    println("$word $name")
}
```

呼叫的時候是依照引數的 **位置** 來映射對應的變數

```
greeting("Peter", "Hello,")
```

這樣的呼叫 name 參數會對應 "Peter" 引數，word 參數會對應到 "Hello, " 引數

kotlin 提供命名參數的對應方式，依照參數的名字來實現參數與引數的對應

```
greeting(word = "Hello, ", name = "Peter")
```

> kotlin 也支援混合使用，條件是 命名參數必須在位置參數之後

動態數量參數

對於參數數量不確定時，稱為 **動態數量參數**，kotlin 中使用 `vararg` 關鍵字就可以將參數宣告為動態數量

```
fun sum(vararg n:Int) {
    var result = 0
    for (a in n) {
        result += a
    }
    return result
}
```

動態數量搭配命名參數

在 java 中如果要動態數量參數和一般參數混合使用，那動態參數一定要擺在最後宣告 在 kotlin 中因為有命名參數，可以無視位置的限制，使用上更彈性

```
fun sum(vararg n: Int, init:Int) {
  return n.sum() + init
}
sum(1,2,3, init = 5)
```

動態參數與陣列

當一個方法參數被宣告為動態參數時，我們傳入一個陣列當作引數，這時候 kotlin 會不知道要把這整個陣列當成 "一個參數"，還是要將這個陣列中的每一個元素當作 "動態參數" 來對應，意思是

```
fun get0(vararg n:Int) {
    println(n[0])
}

val arr = intArrayOf(1,2,3)

get0(arr) // <-- 這個結果會是 1, 還是 [1,2,3] ?
```

Kotlin 規定：陣列預設會作為動態參數的第一個元素來處理

所以答案是 `[1,2,3]` ? 錯! 答案是 `error: type mismatch: inferred tyep is IntArray but Int was expected`

因為宣告時參數為 Int 的變動參數，這時候因為預設將陣列當作動態參數的第一個元素，而 `arr` 的型別是 IntArray 不是 Int，所以型別錯誤!

如果要讓陣列對應動態變數的元素，在呼叫時，引數要標註星號 `*` ，kotlin 會自動解構對應到動態變數中

```
get0(*arr) // 1
```

> 那如果把 get0 設計成 `get00(a:Int, b:Int, c:Int) {}` 然後呼叫 `get00(*arr)` 可以嗎? 答案是... 不行!! 因為 `*` 前綴只能給動態參數使用

參數預設值

kotlin 中宣告方法時可以用 `=` 給定參數預設值，有給定預設值的參數在呼叫時就變成可選填的引數，這在 java 中需要宣告方法多載才能實現

kotlin

```
fun greeting(name:String,word:String = "Hello, ") {
    println("$word $name")
}

greeting("Peter") // "Hello, Peter"
greeting("Peter", "Holy, ") // "Holy, Peter"
```

java 中的方法多載

```
void greeting(String name, String word) {
    System.out.println(word + " " + name);
}
void greeting(String name) {
    greeting(name, "Hello, ");
}
```

方法表達式

方法可以配合 表達式 來使用，達成單行方法宣告

搭配 if 表達式

```
fun abs(x:Int):Int {
    if (x > 0) {
        return x
    } else {
        return -x
    }
}
```

單行宣告

```
fun abs(x:Int):Int = if (x > 0) x else -x
```

省略方法 body

1. 回傳成員

```
class SomeClass {
 var someProperty:Int = 100
 fun getProperty():Int = someProperty
}
```

2. 傳遞參數

```
class SomeClass {
    private fun print(msg:String) {
        println(msg)
    }
    fun doPrint(message:String) = print(message)
}
```

內嵌方法

關鍵字：`inline`

將方法宣告為 `inline` 的影響是，該方法的內容會被覆蓋到呼叫的地方，也就是不會真的產生這個方法

原本是這樣：

```
fun show(par:String) {
    println(par)
}

fun main() {
    show("hello")
}
```

改成這樣：

```
inline fun show(par:String) {
    println(par)
}
```

編譯後會是這樣：

```
fun main() {
  { // 直接把方法內容複製到呼叫的地方
    println("hello")
  }
}
```

這樣除了少一層 stack call 以外 原本 main 方法中的區域資訊都可以存取的到

類別 (Class)

類別亦稱 "型別" 或 "類型"，描述一種資料結構的樣子 程式中看到英文描述 Class、Type 都是類似的概念

宣告格式：

```
[存取範圍] class [類別名稱] {
    // ... 屬性
    // ... 方法
}
```

> 如果一個類別沒有實體內容，kotlin 允許省略 `{}`

- 存取範圍： `public` 、 `internal` 、 `private` 三種

物件 (Object)

物件亦稱 "實例"、"實體"，是指照著類別描述的樣子創造出來的物體 程式中看到英文描述 Object、Instance 都是這個概念

在 java 中需要透過 `new` 關鍵字來創建物件，kotlin 中可以省略 `new` 關鍵字

類別建構子 (constructor)

kotlin 中類別建構子分為主建構子及副建構子

主建構子

主建構子的宣告緊跟在類別宣告之後

- 所有物件在創建時，都必須使用主建構子進行初始化
- 一個類別只能有一個主建構子

宣告格式：

```
class [類別名稱] [存取範圍] constructor(屬性列表) {}
```

- `constructor` 關鍵字：當主建構子前不包含任何存取範圍宣告時，此關鍵字可以省略
- 存取範圍：預設為 `public` (可省略)，並還有 `protected` 、 `internal` 、 `private` 共四種
- 屬性列表：在這裡宣告此類別有哪些屬性，此處宣告的屬性必須加上 `val` 或 `var` 的前綴，如果沒有加上前綴，則該參數會被當成常數存在於初始化後的物件中

範例

```
// 基礎宣告式
class Button(var text:String, val width:Int, val height:Int)

// 給定參數預設值 -> 建構子多載
class Button(var text:String = "Click", val width:Int, val heigh
```

初始化區塊

kotlin 的主建構子沒有 java 中的建構實體區塊，對應的是提供初始化區塊來實現相同功能

不一樣的是一個類別中可以擁有多個初始化區塊，這些初始化區塊的執行順序為由上往下一次執行

```
class Button(var text:String, val width:Int, val height:Int) {
    val area:Int
    init {
        area = width * height
        println("初始化區塊 1")
    }
    init {
        println("初始化區塊 2")
    }
}
```

副建構子

- 副建構子的作用在於 **輔助主建構子**
- 所以 **所有的副建構子最終都必須呼叫主建構子**
- 形成 **間接呼叫主建構子==的使用方式**
- **副建構子可以有 0 個或多個，但是主建構子只能有一個**

宣告方式：

```
constructor(參數列表1):this(參數列表2) {}
```

- 參數列表 1 : 表示此副建構子的參數列表
- 參數列表 2 : 表示呼叫主建構子或其他副建構子

> 副建構子的參數並不是屬性，只有主建構子才能定義屬性，副建構子只是輔助，並不會產生物件 使用副建構子創建最終還是靠主建構子來產生物件

```
class Button(var text:String, val width:Int, val height:Int) {
    // 副建構子 1 呼叫主建構子
    constructor(text:String):this(text,0,0) {}

    // 副建構子 2 呼叫主建構子
    constructor(width:Int, height:Int):this("",width,height) {}

    // 副建構子 3 呼叫 副建構子 1
    constructor():this("Default")
}
```

屬性

來深入思考一下屬性是什麼

在 JavaBean 中，我們一般不會嚴格區分成員和屬性的差別，意思是

```
class Demo {
    private String _alias_;
    public String getName() {
        return _alias_;
    }
    public void setName(String name) {
        _alias_ = name;
    }
}
```

在上述範例中 `_alias_` 是類別成員， 但是對於這個類別的物件來說，能夠操作的屬性是 `name` 而屬性 `name` 對應的訪問器 (getter/setter) 就是 `getName` 和 `setName` 方法

會特別把這件事提出來是因為在 kotlin 中開發者只能定義屬性，不能定義成員

> 由上述例子可以看出，事實上我們並不在意真正成員叫什麼名字，對於物件來說，屬性才是操作標的

在 kotlin 中 **每一個屬性都會自動創建預設的訪問器** 某些情況下，我們需要自定義屬性的訪問器：

- getter 和 setter 必須緊跟在屬性名稱後面
- getter 和 setter 沒有先後順序的要求
- getter 需要回傳屬性值
- setter 會包含一個參數，該參數用來保存傳遞過來的引數

```
class Computer(val disk:Int) {
    var availableSpace:Int = 0
    var usedSpace:Int
        get() {
            return disk - availableSpace
        }
        set(value) {
            availableSpace -= value
        }
}
```

讓我們思考一下這段程式

```
var price:Int = 0
    set(value) {
        if (value < 0) {
            price = 0
        } else {
            price = value
        }
    }
```

價格不能小於零，看起來很正常，問題在哪?

當呼叫 `price = 0` 的時候，stack 就爆了，why? 思考一下

```
設定 `price = 0` -> 呼叫 `price` 的 `set` 方法 ->
判斷 `value` 沒有小於零 -> 執行 `else` 區塊 ->
設定 `price = 0` -> 回到一開始
```

看出來了嗎? 這樣的呼叫會像無限遞迴一樣的呼叫，直到 stack 記憶體被 function call stack 填滿噴出 stackoverflow

隱藏成員

要在 setter 中修改真正的成員值而不是再次使用 setter 方法來賦值，可以呼叫隱藏參數 `field`，設定了 `field` 參數值等於設定了該屬性背後的成員變數值

```
var price:Int = 0
 set(value) {
   if (value < 0) {
     field = 0
   } else {
     field = value
   }
 }
```

這樣就不會造成無限遞迴呼叫的狀況，因為設定不是 price 這個屬性，而是此屬性的隱藏成員

內聯屬性

關鍵字 `inline` 宣告的屬性不會有隱藏成員，這也表示開發者需要自己去實現那個隱藏成員的動作，就像上述 java 範例那樣

```
class Student {
    var _alias_ = ""
    inline var username:String
        get() = _alias_
        set(value) {
            this._alias_ = value
        }
}
```

延遲初始化屬性

在 Kotlin 中，類別定義的所有屬性都必須明顯的被初始化，這個規定可以提醒開法者及時注意屬性是否有正確被初始化，但某些時候會造成開發者必須多寫一些程式碼，例如必須要延遲初始化的屬性，在創建物件時不會帶入主建構子，就必須宣告為可空 (Optional -> `?`)，一旦宣告為可空，之後對於屬性的操作就必須做非空處理

例如 Android，屬性必須在 `onCreate` 後才會被初始化，之前都是空

```
class SomeActivity:Activity {
    var txtTitle:TextView? = null
    override fun onCreate(savedInstanceState:Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_some)

        txtTitle = findViewById(R.id.title)

        txtTitle!!.text = "手動強制非空賦值"
    }
}
```

Kotlin 有提供 `lateinit` 關鍵字來聲明屬性一定會被初始化，只是會延遲，這種屬性有幾個限制：

- 屬性必須是在類別中的成員，不能在主建構子中宣告
- 屬性必須使用預設的訪問器，不支援自定義訪問器
- 屬性必須宣告為 `var`
- 屬性必須宣告為非空
- 屬性必須非基本數據類型

```
class SomeActivity:Activity {
    lateinit var txtTitle:TextView
    override fun onCreate(略) {
        txtTitle = findViewById(R.id.title)
        txtTitle.text = "屬性為非空，無須強轉"
    }
}
```

類別方法

有些方法的使用必須搭配類別

infix

infix 方法是一種 若且為若只有一個參數 的方法 效果是可以像 二元操作子 那樣使用

```
class TeamMember {
    var point:Int = 0
    infix fun addPoint(p:Int) {
        this.point += p
    }
}

fun main() {
    val student = Student()
    student addPoint 2 // 二元操作方式
    println(student.point) // 2
}
```

componentN

是 `operator` 方法的一種特別修飾字 以 `component` 開頭並且以自然數為結尾命名的方法 此類方法沒有任何參數，其主要目的從物件中提取對應的屬性值 常用於物件的解構式中

```
class Student(val name:String) {
    var height:Int = 0
    var weight:Int = 0

    operator fun component1():String = name

    operator fun component2():Int = height

    operator fun component3():Int = weight
}

fun main() {
    val peter = Student("Peter")
    peter.height = 180
    peter.weight = 70
    println(peter.component1()) // "Peter"
    println(peter.component2()) // 180

    // 解構式

    val (name, height, weight) = peter // 依照位置解構
    println(name) // "Peter"
    println(height) // 180
    println(weight) // 70
}
```

內嵌類別 (Nested Class)

定義在類別內部的類別

完整的類別名稱為 "外部類別.內嵌類別"

內嵌類別是靜態存在的，只是邏輯上和外部類別存在一定關聯，但實際上內嵌類別和外部類別是獨立存在的，所以 **內嵌類別無法訪問外部類別的成員**

```
class Outer {
    val outerProperty:String = "外部類屬性"

    class Nested {
        val nestedProperty:String = "內嵌類屬性"

        fun printProperty() {
            // 無法訪問 outerProperty
            println(nestedProperty)
        }
    }
}

fun main() {
    val nested = Outer.Nested()
    nested.printProperty() // "內嵌類屬性"
}
```

內部類別 (Inner Class)

將類別做為另一個類別的成員

修飾子：`inner`

內部類別是外部類別的一個成員，內部類別無法獨立存在，需要透過外部類別的物件才能創建 也因為這樣，所以 **內部類別可以訪問外部類別的屬性**

```
class Outer(private val name:String = "Outer") {
    inner class Inner {
        fun getOuterName() = name
    }
}

fun main() {
    val outer = Outer()
    val inner = outer.Inner() // 需要透過外部類別物件才能創建
    println(inner.getOuterName()) // "Outer"
}
```

單一實例 (Singleton)

kotlin 中叫做 "Object Declaration"

關鍵字：`object`

這種類型的類別只會有單一實例，也就是設計模式中的 Singleton Pattern 此類別實例的創建過程是 thread-safe 的 因為此類別實例只會有一個，使用時 像 靜態類別呼叫就好，不需要在創建類別實例

```
object Singleton {
    private var num:Int = 0
    fun sequence():Int {
        num += 1
        return num
    }
}

fun main() {
    println(Singleton.sequence()) // 1
    println(Singleton.sequence()) // 2
    println(Singleton.sequence()) // 3
}
```

> 這種用法單純是 "像" 靜態方法呼叫而已，實際上差異很大，Kotlin 只是隱藏 Singleton Pattern 的執行過程並簡化呼叫而已 實際上在 Kotlin 語法中，沒有靜態成員這種東西

伴生實例 (Companion)

在類別中的 **單一實例** 成員 **一個類別只能有一個伴生實例**

修飾子：`companion`

原理同 **單一實例** 類別，使用上加上了 `companion` 修飾子，就可以像呼叫類別靜態方法一樣的使用

> 再次注意，Kotlin 中沒有靜態方法，沒有靜態方法，沒有靜態方法
> 這裡說的只是 使用上的類似

```
class AlertDialog(var title:String, var message:String) {
    companion object LikeStatic {
        fun method() {}
    }
}

fun main() {
    AlertDialog.LikeStatic.method() // 通過伴生實例呼叫方法
    AlertDialog.method() // 省略伴生實例名稱，外部類別直接呼叫伴生實
}
```

companion object 不一定需要給名字，使用上就像 `static` 一樣

```
class AlertDialog(var title:String, var message:String) {
    companion object {
        fun method() {}
    }
}

fun main() {
    AlertDialog.method()
}
```

物件表達式 (Object Declaration)

類似創建匿名類別物件的方法

1. 使用 `object` 關鍵字

`object` 關鍵字是創建單一實例的方法，也能用來創建匿名類別的物件

```
fun main() {
    val obj = object {
        fun hello() = "Hello"
        var property = "Oops"
    }
    println("hello:${obj.hello()}, property:${obj.property}")
    // "hello:hello, property:Oops"
}
```

1. 繼承或實現

```
interface Valuable {
    fun getPrice():Int
}
abstract class Item {
    abstract var group:String
}

fun main() {
    val obj = object : Item(), Valuable {
         override fun getPrice(): Int = 0
         override var group: String = "SomeGroup"
    }
}
```

匿名物件存取範圍限制在 本地作用域(`{}` 之中) 或 物件私有域 之中

也就是說，在存取範圍內的程式，或者將匿名物件當作 `private` 方法回傳值的時候，是可以存取匿名物件中的屬性或者方法的

相對的，如果該匿名物件被當作 `public` 方法的回傳值，那在第 1. 種情況，回傳的型別會是 `Any` ，而第 2. 種情況回傳的會是 `Item, Valuable` 的型別，只能使用父類別或介面的方法和屬性

```
open class Parent {
    var y:String = "y"
}
class Child {
    private fun privateMethod() = object : Parent() {
        val x:String = "x"
    }

    fun publicMethod() = object : Parent() {
        val x:String = "x"
    }

    fun test() {
        val priObj = privateMethod()
        println(priObj.x) // 可以存取匿名物件的屬性
        println(priObj.y) // 也可存取匿名物件父類別的屬性

        val pubObj = publicMethod()
        println(pubObj.y) // 只能存取到父類別的屬性
    }
}
```

使用上，匿名物件可以存取該作用域內定義的成員

```
fun test() {
    var x = 10
    var obj = object {
        var y = x * 10
    }
}
```

數據物件 (Data Object)

> 貧血模型 (Anemic Domain Model) 是一種除了 getter 和 setter 以外沒有別的方法的類別結構 在 Java 中常用的 JavaBean 就是一種典型的貧血模型 也類似 Lombok 中的 @Data 標註

Kotlin 中將類別宣告為 `data` 型類別，並有一些條件

- 主建構子至少需要一個屬性
- 主建構子只能包含屬性，不能包含參數 -> 一定要使用 `var` 或 `val` 修飾
- 類別宣告不能有 `abstract` 、 `open` 、 `sealed` 、 `inner` 修飾子
- 不可以覆寫 `copy()` 及 `componentN()` 方法

kotlin 會自動覆蓋以下方法：

- `equals()`：會自動比較屬性內容，而不是參考位置
- `hashCode()`：雜湊值也會依照屬性內容計算
- `toString()`：自動套用 `類別名稱(屬性名稱=屬性值)` 的文字模板
- `componentN()`：自動產生依照建構順序的解構方法
- `copy()`：淺層複製物件

注意細節

1. 基本上屬性都包含在主建構子中，如果希望某些屬性能夠被排除，那就得將屬性宣告在類別內部

```
data class SomeData(val name:String, val age:Int) {
    var size:Int = 0 // 會被排除計算
}
```

2. 如果手動實現了自動覆蓋的方法，或者父類別對這些方法宣告了 `final`，則不會覆蓋
3. 如果父類別中聲明了 final 的 componentN 解構方法，或者該方法的簽名與子類別不兼容，編譯器會直接報錯
4. 不能繼承有自定義 `copy()` 方法的父類別

列舉 (enum)

修飾子：`enum`

星期範例：

```
enum class WeekDay(val abbr:String) {
    Monday("Mon"), Tuesday("Tue"), Wednesday("Wed"), Thursday("T
    Friday("Fri"), Saturday("Sat"), Sunday("Sun");

    fun isWeekEnd():Boolean = this == Saturday && this == Sunday
}
```

> ## 重要!
>
> 如果在列舉值之下還有其他部分程式，像是方法或其他宣告，那列舉值結束需要以分號 `;` 結尾 這可能是整個 kotlin 語言中唯一強制需要加上分號的地方了

使用方法：

1. 直接使用

```
val monday = WeekDay.Monday
```

2. 透過隱藏的 name 屬性 (name 屬性和列舉成員值相同)

```
val mondayByName = WeekDay.valueOf("Monday")
```

使用這種方法取得，如果帶入的字面值沒有對應的列舉值，則會拋出 java.lang.IllegalArgumentException: No enum constant

3. 透過位置取得

```
val mondayByOrder = WeekDay.values()[0]
```

列舉值方法

列舉值的方法會寫在每一個列舉對象的閉包內

```
enum class WeekEnd(val abbr:String) {
    Saturday("Sat") {
        override fun chineseDesc() = "星期六"
    },
    Sunday("Sun") {
        override fun chineseDesc() = "星期日"
    };

    abstract fun chineseDesc():String // 此類別的實例(列舉對象)都要
}
```

密封

修飾子：`sealed`

密封類別個人解讀是一種強化列舉的類別，在列舉中，所有列舉值的型別都是該列舉類別，所有列舉對象具有相同的屬性，相同的方法

但是在密封類別中，每一個密封對象可以有不同的屬性和不同的方法，因為其實他們都是不同的類別，只是有共同的父類別而已

使用上 `open` 和 `sealed` 的差異是 密封類別的主建構子是私有的，所以無法在其他地方直接生成密封類別的物件 但是 `sealed` 的 **內嵌子類別** 就沒有這個限制

```
sealed class Event {
    class Send(val target:String):Event()
    class Receive(val from:String):Event()
}
```

- `Send` 有 `target` 的屬性
- `Receive` 有 `from` 屬性

使用上相當於建立一個 Event 子類別的實例

```
val sendToJhon = Event.Send(target = "John")
val receiveFromTom = Event.Receive(from = "Tom")
val event = Event() // 報錯! 不能直接產生密封類別的物件
```

如果說只需要一個純粹的子類別不含屬性並且希望像 enum 那樣使用，可以搭配 `object` 關鍵字

```
sealed class Event {
    class Send(val target:String):Event()
    class Receive(val from:String):Event()
    object OnTheWay : Event()
}
```

這樣在取得 `Event.OnTheWay` 的時候就會取得單一實例的物件，不會每一次使用都創建新物件出來

使用上搭配 `when` 可以用 `is` 來判定型別，並且在每一個區塊內會自動轉型

```
fun main() {
      val sendToJhon = Event.Send(target = "John")
      val receiveFromTom = Event.Receive(from = "Tom")
      val transmitting = Event.OnTheWay

      val listEvents = listOf(sendToJhon, receiveFromTom, transmi

      when(val obj:Event = listEvents.random()) {
          // 會將 obj 自動轉型成 Event.Send，所以可以存取 target 屬
          is Event.Send -> println(obj.target)
         // 會將 obj 自動轉型成 Event.Send，所以可以存取 from 屬性
         is Event.Receive -> println(obj.from)
         // 單一實例不需要用 is 型別判斷，直接等於判斷即可
         Event.OnTheWay -> println("event on the way")
      }
}
```

繼承

Kotlin 中預設所有類別都會被冠上 `final` 的關鍵字以後編譯成 class 檔 所以如果要把類別開放給其他類別繼承，要在類別宣告前加上 `open` 修飾子 同樣的邏輯也用於方法和屬性，預設都是 `final`，如果要開放則要增加 `open` 修飾子

- 如果要繼承某個類別，要在類別宣告最後方加上冒號 `:` 並指定要繼承的類別

```
open class Super {}
class Sub:Super {}
```

> Kotlin 中只能繼承一個類別 (和 Java 相同)

- 如果父類別有自定義建構子，那子類別宣告時也要指定使用父類別的哪一個建構子 子類別建構子中的屬性不需要再宣告 `val` 或 `var`

```
open class Super(val name:String = "")
class Sub(n:String):Super(n)
class Sub2:Super() // 因為父類別 name 屬性有給預設，可以呼叫無參數建
```

- 透過建構子實現繼承

```
class Sub3:Super {
    constructor(n:String) : super(n)
}
```

覆寫屬性

覆寫屬性可以把屬性從 `var` 覆寫為 `val`，但是反過來不行

```
open class View(val width:Int, val height:Int) {
    open var size:Int = 0
}

class SmallView(width:Int, height:Int):View(width, height) {
    override var size:Int = width * height
}

class largeView(width:Int, height:Int):View(width, height) {
    override var size:Int = width * height
        get() = width * width // 宣告只有 get()，表示此屬性從 var
}
```

內部類訪問父類別

子類別的內部類別要訪問外部類別的父類別屬性，要使用 `super@外部類別.父類別屬性`

```
open class View(val width:Int, val height:Int) {
    open var size:Int = 0
}

class SmallView(width:Int, height:Int):View(width,height) {
    override var size:Int = width * height

    inner class Painter {
        fun sizeInOuterClass():Int = size // 拿到的會是 SmallView
        fun sizeInSuperClass():Int = super@SmallView.size
    }
}
```

抽象類別

修飾子：`abstract`

組成：

- 抽象屬性
- 抽象方法
- 實體屬性
- 實體方法

繼承規則

- 如果實體類別繼承抽象類別，實體類別必須實現所有抽象類別的屬性和方法
- 如果抽象類別繼承抽象類別，則可以選擇想要實現的屬性和方法，也可以都不要

介面

關鍵字：`interface`

組成：

- 實體方法
- 抽象方法
- 抽象屬性
  - 聲明屬性而不進行初始化操作
  - 聲明屬性後提供自定義 getter，因為抽象屬性不包含隱藏成員，所以不能使用 `field`

```
interface OnTouchListener {
  var target:View
  val description:String
      get() = "a touch listener"
}
```

> Kotlin 一個類別可以實現多個介面 (也和 Java 一樣)

實現衝突

當一個類別實現的多個介面有實現相同命名的方法 **子類別必須實現該方法** 可以使用 `super<父類別型別>.方法()` 在實現方法的區塊內分別呼叫不同覆類別的該方法

```
interface A {
    fun call() {
        println("call in A")
    }
}

interface B {
    fun call() {
        println("call in B")
    }
}

class Concrete : A, B {
    override fun call() {
        super<A>.call()
        super<B>.call()
        println("call in Concrete")
    }
}

fun main() {
    val con = Concrete()
    con.call()
    /**
    call in A
    call in B
    call in Concrete
    */
}
```

套件 (Package)

預設導入的套件

- `kotlin.*`
- `kotlin.annotation.*`
- `kotlin.collections.*`
- `kotlin.comparisons.*`
- `kotlin.io.*`
- `kotlin.ranges.*`
- `kotlin.sequences.*`
- `kotlin.text.*`

不同平台可能會額外導入其他的套件

- JVM：`java.lang.*`、`kotlin.jvm.*`
- JS：`kotlin.js.*`

修改套件名稱

可以使用 `as` 來修改套件在本地端的名稱

```
val bar = foo.Bar()
```

等同於

```
import foo.Bar
val bar = Bar()
```

等同於

```
import foo.Bar as FooBar
val bar = FooBar()
```

存取修飾子

用於類別、介面、建構子、方法、屬性

- `public`：預設，可省略，被修飾的對象全域可存取
- `private`：被修飾的對象只能在類別內部存取

- `protected`：被修飾的對象只能在類別內部或者子類別中存取
- `internal`：被修飾的對象只有在類別所在的 "模組" 中可存取

> ## Kotlin 模組
>
> 模組是一個編譯單元，表示放在一起編譯的原始碼，一般來說，實際項目打包輸出的 jar 檔就算是一個模組

建構子上的存取修飾子

預設建構子的存取修飾子是 `public`

> 只要能存取到該類別就能建構該類別的物件

如果要修改建構子的預設存取級別，則要使用 `constructor` 關鍵字 (預設省略) 來宣告主建構子

```
class A private constructor(desc:String) {}
```

這種情況下可以使用伴生實例來創建類別物件

```
class A private constructor(desc:String) {
    private constructor():this("default")
    companion object {
        fun create():A = A()
    }
}

fun main() {
    val a:A = A.create()
}
```

類別上的存取修飾子

在類別成員上的存取修飾子是具有 **傳遞性** 的，意思是如果被覆寫的成員用 `xxx(public/protected/private/internal)` 修飾，則覆寫後的成員預設也是由 `xxx` 修飾

其他修飾子

open / final

`open` 和 `final` 是互斥的修飾子

- `open` 修飾類別表示該類別可以被繼承，`final` 修飾類別表示該類別不可以被繼承，具體類別預設是 `final`，抽象類別預設是 `open`
- `final` 不可使用在介面上，介面預設是 `open`，所以也不需要多寫 `open`
- `open` 修飾方法或屬性表示該方法或屬性可以被覆寫，`final` 修飾方法或屬性表示該方法或屬性不能被覆寫，預設方法或屬性都是 `final`，而抽象方法或屬性預設是 `open`

override

只能用於子類別覆寫父類別的方法或屬性上，只能覆寫 `open` 的成員

目標

... 規劃中

Let's go

1. IntelliJ IDEA 創建 java + springboot 專案
2. 引入 kotlin plugin
    - kotlin jvm plugin
3. 引入必要依賴
    - kotlin-stdlib

... 規劃中

重要的延伸

- 例外處理
- 執行緒
- 泛型
- 擴展與委託
- 標註與反射

其他延伸

- DSL
- 和 Java 之間互動
- 標註與反射
- IO操作

基礎宣告

泛型符號

一般使用的泛型符號為大寫英文字母，其中某些字母為使用上的常例

- `T` : 表示 Type : 通用的型別符號
- `K` : 表示 Key : 用於表達該型別為鍵值使用型別
- `V` : 表示 Value : 用於表達該型別為內容值使用型別
- `E` : 表示 Exception : 用於表示例外的型別

泛型方法

在 `fun` 關鍵字後方以 `<T>` 宣告方法使用泛型 後方方法宣告鐘就可以使用 `T` 來當作一種型別

```
fun <T> echo(t:T):T {
    return t
}
```

泛型類別

在類別名稱後面加上 `<T>` 來宣告類別使用泛型

> Kotlin 不支援泛型原始型別直接使用

```
open class SomeClass<T>(obj:T)
```

繼承

如果繼承一個泛型類別，子類別需要明確指定父類別泛型的型別，而不能沿用父類別的泛型參數

```
open class Super<T>(obj:T)

class Sub(obj:Int):Super<Int>(obj)
```

泛型限制

指定泛型型別的上界，有助於對泛型物件操作的掌握性

```
fun <T> sum(first:T, second:T) = first + second
```

這樣的方法會導致意料之外的結果，如果 T 是數值型別物件，那可能會照預期結果做相加的動作，但是如果是其他類型物件，則結果就無法預期，這種情況可以對泛型型別 `T` 做型別上界的限制

```
fun <T:Number> sum(first:T, second:T) = first + second
```

這樣調整可以限制泛型型別 `T` 的上界一定要是 `Number` 型別，這樣的設定就能保證此方法的結果和預料結果相同，當傳入非數值物件時，編譯器會報錯

多個限制

使用 `where` 宣告來指定多個限制

`where` 需要放在類別或方法的宣告的地方，多個限制以逗號 `,` 做分隔

- 每一個限制都必須包含泛型型別
- 因為 kotlin 單繼承特性，多個限制型別中只能有一個是具體類別

```
fun <T> remove(Collection: T, item: T) where T:MutableIterable<T

abstract class MyList<T>:Collection<T> where T:MutableIterable<T
```

預設限制

泛型型別 `T` 在 kotlin 中預設以 `Any?` 為型別上界，如果沒有特別指明上界了話在使用時需要做可空 `?` 處理

```
fun <T> eq(first:T,second:T):Boolean {
    return first?.equals(second)
}
```

可空處理在非必要的時候用起來很不方便，可以強制將泛型上界定為 `Any`

```
fun <T:Any> eq(first:T,second:T):Boolean {
    return first.equals(second)
}
```

> 在 Java 這個預設限制是 `Object`

轉型

kotlin 泛型預設是不允許向上轉型的，例如

```
open class Super
class Sub:Super()

class Holder<T>(var obj:T)

fun main() {
    var subHolder:Holder<Sub> = Holder<Sub>(Sub())
    var supHolder:Holder<Super> = Holder<Super>(Super())
    supHolder = subHolder // 向上轉型 -> 編譯錯誤!
}
```

向上轉型 (Covariant) -> 協變

如果 Sub 是 Super 的子類別，那 Sub 的型別物件轉換成 Super 型別，這種特性稱為 "向上轉型"

在宣告時，需要在泛型符號前聲明 `out` 關鍵字

```
class CovariantHolder<out T>(var obj:T)

fun main() {
    val strHolder = CovariantHolder<String>("super")
    val anyHolder:CovariantHolder<Any> = strHolder
    // 自動將 String 向上轉為 Any, 不會爆錯
}
```

> 向上轉型轉換後只能當做 取值 物件，即只能作為方法回傳值或唯讀屬性 例如上面 `anyHolder.obj = Date()` 這種操作是不允許的 此例中因為 anyHolder 的 `obj` 實體其實是 strHolder 宣告的 String 型別 雖然 anyHolder 以 `Any` 型別轉換，理論上看起來可以 但是實際上賦值的時候因為 String 和 Date 型別並不能互轉 所以會拋出錯誤 因為如此，所以 kotlin 不允許向上轉型轉換後賦值操作，只允許 取值操作**(out)**

向下轉型 (Contravarint) -> 逆變

如果 Sub 是 Super 的子類別，那 Super 的型別物件轉換成 Sub 型別，這種特性稱為 "向下轉型"

在宣告時，需要在泛型符號前聲明 `in` 關鍵字

和向上轉型相反，向下轉型 只允許賦值操作**(in)**，即只能當作引數輸入到其他地方

```
class ContravarintHolder<in T>(obj:T) {
    fun test(param:T) {
        println(param)
    }
}

fun main() {
    var any = ContravarintHolder<Any>(10) // Any 型別，值為 10
    var str:ContravarintHolder<String> = any // 因為 String 是 An
    str.test("hello") // 逆變型別只允許賦值操作
}
```

轉型邊界

如果同一個參數同時需要有賦值和取值的功能，那怎麼辦? 以上的泛型型別宣告都是在類別宣告的時候就指定了，影響範圍是整個類別的成員 = 無解

要實現可以輸入也輸出，只能縮小轉型範圍，例如縮小到只存在於方法的參數中

```
fun covariant(holder: Holder<out Number>) {
    val value = holder.obj // 可以! 輸出屬性
    holder.obj = 5 // 噴錯! out 只能用於輸出
}

fun contravarint(holder: Holder<in Number>) {
    holder.obj = 5 // 可以! 輸入屬性 -> setter 的引數
    val any:Any? = holder.obj // 因為 in Number 的宣告表示，輸入進
}
```

泛型擦除

和 Java 一樣，泛型聲明只存在於編譯期，編譯後泛型的型別不會保留泛型的類型在編譯後的 class 檔中

這表示無法將泛型型別當作普通類別來使用，例如 `obj is Holder<Int>` ，後方那個 `Holder<T>` 的 `<T>` 的型別 `<Int>` 在編譯後並不會保存

也無法用泛型型別來創建物件

```
fun <T> newIt(t:T):T {
    return T::class.java.newInstance() // nope! 編譯器錯誤
}
```

要解決這種問題處理方式一般都是在使用時也帶入一個類別型別的參數來做判斷

```
fun <T> fillList(list:MutableList<T>, size:Int, clazz: Class<T>)
    for (i in 1..size) {
        val obj = clazz.newInstance()
        list.add(obj)
    }
}
```

那有沒有什麼辦法可以讓方法保留泛型資訊

有!

1. 使用 `inline` 內嵌方法，原理是內嵌方法本體會被覆蓋到呼叫的位置，所以原本輸入泛型引數的地方會被 "真正的引數" 代替，所以拿到會是 "真正引數的型別"

2. 對於泛型型別 `T` 增加 `reified` 修飾子，告訴編譯器 用 **"真正的類別參數型別"** 來替代這個泛型型別

`reified` 只能搭配 `inline` 使用

```
inline fun <reified T> fillList(list:MutableList<T>, size:Int) {
    for (i in 1..size) {
        val obj = T::class.java.newInstance() // 用真實類型產生實
        list.add(obj)
    }
}
```

範例

使用泛型實現 Dao 架構

```kotlin
open class Entity() {
    var id:Long?
    var createdAt:Data?
    var updateAt:Date?
}

class Product:Entity() {
    var code:String?
    var name:String?
    var price:BigDecimal?
}

class Stock:Entity() {
    var name:String?
    var products:List<Product> = mutableListOf()
}

interface Dao<T> {
    fun save(t:T):Long
    fun get(id:Long):T?
}

abstract class BaseDao<T:Entity>:Dao<T> {
    var table = mutableMapOf<Long,T>()

    override fun save(t:T):Long {
        t.id = System.currentTimeMillis()
        t.createdAt = Date()
        t.updateAt = t.createAt
        table[t.id!!] = t
        return t.id!!
    }

    override fun get(id:Long):T? {
        return table[id]
    }
}

class ProductDao:BaseDao<Product>()
class StockDao:BaseDao<Stock>()

fun main() {
    val productDao = ProductDao()
    val product = Product().apply {
        code = "B00001"
        name = "Kotlin"
        price = BigDecimal("60")
    }
```

```
    val pid = productDao.save(product)

    varl productRecord = productDao.get(pid)

    val stockDao = StockDao()
    val sotck = Stock().apply {
        name = "Storage 1"
        products = mutableListOf(product)
    }
    val sid = stockDao.save(stock)
    val stockRecord = stockDao.get(sid)
}
```

擴展 (Extension)

為既有類別添加新的方法或屬性

> 可以擴展 Kotlin 原生的類別，也可以擴展 Java 的類別 實際上很多 Kotlin 中的 lib 或語法糖都是擴展 Java 類別得來的

擴展方法

語法：`fun 類別名稱.擴展方法名稱(參數):回傳型別`

範例：對 Num 型別擴展 add 方法

```
class Num(var x:Int)

fun Num.add(var y:Int) {
    this.x += y
}

fun main() {
    var n = Num(10)
    n.add(4)
    println(n.x) // 14
}
```

> 如果擴展的方法名稱和原本類別中的名稱衝突(相同)，則呼叫的時候 一定是呼叫類別中定義的方法，不會調用擴展方法

擴展屬性

擴展屬性並不是真的在底層類別中增加一個屬性，而是以增加特殊的 getter/setter 方法來實現

已經定義好的類別本身沒有額外空間來儲存新的屬性，所以擴展屬性不能有初始值和隱藏變數(`field`)

語法：`val 類別名稱.屬性名稱:屬性型別`

範例：為 List 添加一個 second 屬性取得列表第二個值 (索引值為 1)

```
val <T> List<T>.second:T
    get() = [1]

fun main() {
    val list:List<Int> = listOf(1,2,3)
    println(list.second) // 2
}
```

擴展伴生實例

語法：`fun 類別名稱.Companion.伴生實例名稱:回傳值型別`

為 String 類別添加 `random` 伴生實例

```
fun String.Companion.random():String = UUID.randomUUID().toStrin

fun main() {
    println(String.random()) // UUID
}
```

擴展可空類型

如果把擴展功能放在可空類型上，則在擴展功能中藥添加判斷當接收者為空時的邏輯，這樣在實際使用時就不用為擴展功能強制加上安全操作：`?`

語法：`fun 類別名稱?.擴展方法(參數):回傳型別`

範例：如果要幫字串添加一個 `md5` 的屬性，但是當字串是空值時就不要處理，回傳空字串

```
val String?.md5: String
    get() {
        if (this == null) {
            return ""
        }
        return BigInteger(1, MessageDigest.getInstance("MD5").di
     }


fun main() {
    val s = "123"
    println(s.md5) // "202cb962ac59075b964b07152d234b70"
    val s2:String? = null
    println(s2.md5) // ""
}
```

靜態綁定

將擴展方法綁定到類別的過程是採用所謂 **靜態綁定** 的方式 意思是在呼叫擴展方法時，會呼叫的是使用時的 **聲明類型**，而不是運行時實際物件的類協

```
open class A
class B:A()
fun A.test() = println("test in A")
fun B.test() = println("test in B")
fun call(a:A) = a.test()
fun main() {
  val b:B = B()
  call(b) //  "test in A" : 引數是 B 類型物件，呼叫的卻是 A 類型的
  // 因為 call 方法宣告時聲明的是 A 類型，所以會呼叫 A 的擴展方法
   b.test() // "test in B"
}
```

代理

方法代理

代理是一種 Design Pattern，Kotlin 有特殊語法可以省去撰寫代理模式那一堆的程式碼

範例：代理模式

```
interface Base {
    fun log(message:String)
    fun isEnabled():Boolean
}
class BaseImpl:Base {
    override fun log(message:String) = println(message)
    override fun isEnabled() = true
}
class Delegation(val base:Base):Base {
    override fun log(message:String) {
        base.log(message)
    }
    override fun isEnabled() = base.isEnabled()
}

fun main() {
    val baseDelegate = Delegation(BaseImpl())
    baseDelegate.log("test") // "test"
}
```

這就是一個很簡單的代理模式，實際運用上會有個小問題，就是當 Base 介面方法很多的時候，**這個代理也會需要一一實現這些方法**，這些程式碼雖然意義明確但是很瑣碎，kotlin 提供了一種語法可以讓代理只需要覆寫指定的方法，不需要全部方法都覆寫

語法：`代理介面名稱 by 代理的實例`

```
class Delegation(base:Base) Base by base {
    override fun log(message:String) {
        println("override log")
        base.log(message)
    }
    // 其他方法如果沒有特殊處理可以不寫，Kotlin 編譯器做掉
}
```

屬性代理

除了呼叫方法可以製作代理，屬性也能有代理

底層實現方法原理是將屬性的 getter/setter 呼叫過程讓代理的 `getValue()/setValue()` 來完成

```
operator fun getValue(thisRef:Any?, prop:KProperty<*>):屬性型別 {
    return 屬性代理物件
}
```

- `thisRef:Any?` ：表示這個被代理的對象實體
- `prop` ：表示對代理屬性的使用
- 回傳值型別必須和被代理的屬性型別一致或為父類別

```
operator fun setValue(thisRef:Any?, prop:KProperty<*>, value:屬性
    屬性代理物件 = value
}
```

- `value` ：setter 方法中的 value 是屬性被賦值時傳遞的的真實值
- setter 的 value 型別必須和被代理的屬性型別一致或為子類別

語法：`var/val 屬性名稱:屬性類型 by 代理對象`

1. 創建代理對象類別
2. 使用 `by` 語法

```
class IntDelegate {
    private var logLevel:Int = -1

    operator fun getValue(thisRef:Any?, prop:KProperty<*>):Int {
        // 回傳值型別可以是 Int 的父類別
        return logLevel
    }

    operator fun setValue(thisRef:Any?, prop:KProperty<*>, value
        // 回傳值型別可以是 Int 的子類別
        logLevel = value
    }
}

class Logger {
    var intLogger:Int by IntDelegate() // 宣告使用 IntDelegate 來
    intLogger.logLevel = 3 // 呼叫到 IntDelegate 的 setValue
    pirintln(intLogger.logLevel) // 呼叫 IntDelegate 的 getValue
}
```

Kotlin 內建屬性代理

惰性初始 (lazy initial)

Kotlin 中的非空屬性初始值是一種立刻初始的方式，當物件創建完成後該物件的初始值也給定了

惰性初始可以讓該屬性的初始過程從創建時被延遲到 第一次使用時 才初始

語法：`val 屬性名稱:屬性類型 by lazy`

使用 Lazy 類型作為屬性委託，就可以達成惰性初始的效果

```
class A {
    val lazyProp:Long by lazy {
        println("Compute...")
        System.currentTimeMillis() // lambda 不需寫 return
    }
}
fun main() {
    println("main start")
    val e:E = E()
    println("E initialized")
    println("access E's lazyProp:${e.lazyProp}")
    println("access E's lazyProp:${e.lazyProp}")
}
/**
main start
E initialized
Computer
access E's lazyProp:1634881534943
access E's lazyProp:1634881534943
*/
```

可以看到當物件 e 初始化的時候並沒有初始 `lazyProp` 屬性 而是在第一次呼叫 access 之前才做初始動作打印出 `Compute...` 之後第二次 access 就直接回傳屬性值，沒有在進行重新初始的動作 (沒有印出 `Compute...`)

> 惰性初始因為沒有 setter 方法，所以只能用於取值，不能用賦值操作

觀察者 (observer)

觀察屬性值變化的情況，並在變化時執行代理定義的操作

語法：`var/val 屬性名稱:屬性類型 by Delegates.observable("屬性初始值") { prop, old, new -> ()->Unit }`

- `prop` ：被代理的屬性物件
- `old` ：賦值操作前的值
- `new` ：正在被賦予的新值

```
class User {
    var name:String by Delegates.observable("初始值") { prop, old
        if (old != new) {
            println("from $old to $new")
        }
    }
}

fun main() {
    val user = User()
    println(user.name) // "初始值"

    user.name = "John" // "from 初始值 to John"
    println(user.name) // "John"

    user.name = "John" // 雖然賦值但是沒改變內容 (old == new) -> 不
    println(user.name) // "John"
}
```

觀察並判斷是否允許操作 (vetoable)

語法：`var/val 屬性名稱:屬性類型 by Delegates.vetoable("屬性初始值") { prop, old, new -> ()->Boolean }`

當屬性值變化時，回傳 `true` 表示允許操作成功，回傳 `false` 則表示操作失敗，屬性值不變

```
class User {
    var age:Int by Delegates.vetoable(0) { prop, old, new ->
        new < 20
    }
}

fun main() {
    val user = User()
    println(user.age) // 0
    user.age = 10
    println(user.age) // 10
    user.age = 21
    println(user.age) // 10 : 因為回傳 false, 賦值操作失敗
}
```

非空代理 (notNull)

可以允許屬性非空直到 使用前才進行初始

> 和惰性初始的差異是，惰性初始化後，就不能再改變，而非空可以

語法：`var 屬性名稱:屬性類型 by Delegates.notNull<屬性類型>()`

```
class User {
    var notNullStr:String by Delegates.notNull<String>()
}

fun main() {
    val user = User()
    // println(user.notNullStr)  // 在初始化之前呼叫 getter 會噴 I
    user.notNullStr = "initial"
    println(user.notNullStr) // "initial"
    user.notNullStr = "after"
    println(user.notNullStr) // "after"
}
```

映射代理 (map)

可以將類別中的屬性全部保存到 Map 物件中，於是存取屬性就變成存取 Map 裡面的鍵值組，但是操作起來卻是像屬性一樣的操做

語法：`var 屬性名稱:屬性類型 by 映射變數名稱`

```
class Person(map: Map<String, Any?>) {
    val name:String by map
    val age:Int by map
}

fun main() {
    val person = Person(mapOf(
        "name" to "John",
        "age" to 13
    )) // 用 map 當引數傳入
    println(person.name) // 使用屬性存取 map["name"]
    println(person.age) // 使用屬性存取 map["age"]
}
```

High Order Function

一句話：將方法的 **格式** 視為一種類別

方法的格式

方法的參數型別組合加上回傳值的型別組合為方法的 **格式**

例如：

```
fun plus(x:Int,y:Int):Int = x + y
```

就是一個輸入兩個 Int 並回傳一個 Int 類型的方法

而 **輸入兩個 Int 並回傳一個 Int** 就是此方法的型別

所以上述的加法和以下的減法都屬於同一種型別

```
fun minus(x:Int, y:Int):Int = x - y
```

如果方法的格式可以被視為一種類別，那類別可以怎樣被使用，方法也可以那樣使用

> 方法格式視為類別是在執行時期動態編譯

方法類別

語法：

- 宣告：`(參數組合)->回傳值類型`
- 實體：關鍵字 `fun` 取代方法名稱，剩下如一般方法實現方法內容

```
var plus:(Int, Int)->Int = fun(x:Int, y:Int):Int {
  return x + y
}
```

方法參數

既然方法可以是一種類別，那就可以當作另一個方法的參數

```
fun apply100(f:(Int)->Int):Int {
    return f(100)
}

fun main() {
    val inc = fun(x:Int):Int = x+1
    val dec = fun(x:Int):Int = x-1
    println(apply100(inc)) // 101
    println(apply100(dec)) // 99
}
```

方法回傳

既然方法可以是一種類別，也能當作回傳值

回傳方法時，如果是該方法已經賦予給某個變數，則回傳該變數即可，如果是回傳方法 **實體**，那要使用方法參考子 `::` 來回傳

```
fun operation(op:String):(Int,Int)->Int {
    fun plus(x:Int, y:Int) = x + y
    val minus = fun(x:Int, y:Int) = x - y

    return if(op == "+") ::plus else minus
    // plus 是方法實體，需使用 :: 回傳
    // minus 已經變數化，可以直接回傳
}

fun main() {
    val add = operation("+")
    val subtract = operation("-")
    println(add(1,2)) // 3
    println(subtract(4,3)) // 1
}
```

方法表達式

lambda 表達式

- 回傳值就是最後一行的執行結果，不需要再寫 `return`
- 無法聲明回傳型別
- kotlin 支援自動執行 lambda，意思是可以同時宣告並呼叫

```
val result = {x:Int,y:Int -> x+y}(1,2) // 3
```

語法：

```
{ 參數列表 ->
    任意執行語句
}
```

範例：

```
val sum = { x:Int, y:Int -> x + y }
```

等同於

```
val sum = fun(x:Int, y:Int):Int {
    return x + y
}
```

隱藏參數 `it`

如果 lambda 只包含一個參數，則參數聲明本身可以被省略，此時 lambda 會生成隱藏參數 `it` 可以讓內部方法本體使用

```
fun main() {
    val square:(Int)->Int = { it * it }
    println(square(5)) // 25
}
```

最末參數

如果方法的最後一個參數是方法類型，kotlin 允許用 lambda 表達式提取到參數列表的小括號之外

```
fun lastLambda(i:Int, lamb:(x:Int)->Int):Int = lamb(i)

fun main() {
    val result = lastLambda(2) {
        it*it
    }
    println(result) // 4
}
```

lambda 中手動回傳值

lambda 表達式預設式回傳最後一行程式的執行結果，如果要手動回傳(`return`)，因為在 "使用情境" 下，lambda 的上下文並不值觀，所以需要標明 返回的作用域

```
fun apply(list:List<Int>, lambda:(Int)->String) {
    for (item in list) {
        println("將 item:$item 當引數帶入 lambda:${lambda(item)}"
    }
}

fun main() {
    apply(listOf(1,2,3)) {
        if (it % 2 == 0) {
            return@apply "偶數" //特別標註是 return@apply 這個作用
        }
        return@apply "奇數" //特別標註是 return@apply 這個作用域，
    }
    /**
    將 item:1 當引數帶入 lambda:奇數
    將 item:2 當引數帶入 lambda:偶數
    將 item:3 當引數帶入 lambda:奇數
    */
}
```

再談內嵌方法

不參與內嵌的參數

內嵌方法中，如果該方法有方法類型參數，但是 不希望該參數被內嵌到使用的地方，希望該方法能夠獨立出來，可以使用修飾子 `noinline`

```
inline fun apply(list:List<Int>, noinline lambda:(Int)->String){
}
```

返回

因為內嵌方法會將整個方法本體覆蓋到呼叫的地方

這時候如果方法本體內有 `return` 的關鍵字，那呼叫的主程式將會被返回

```
inline fun apply(list:List<Int>, lambda:(Int)->String) {
    for (item in list) {
        println("將 item:$item 當引數帶入 lambda:${lambda(item)}"
    }
}

fun main() {
    apply(listOf(1,2,3)) {
        if (it % 2 == 0) {
            println("偶數")
            return
        }
        println("奇數")
        return
    }
    /**
    "奇數"
    */
}
```

第一次進入 lambda 的時候印出 "奇數" 後就直接把 main 返回了 如果不希望有這種誤用發生 可以用 `crossinline` 來修飾方法類型變數，這樣就不允許再方法本體內撰寫 `return` 關鍵字，編譯器會值報錯

```
inline fun apply(list:List<Int>, crollinline lambda:(Int)->Strin
    for (item in list) {
        println("將 item:$item 當引數帶入 lambda:${lambda(item)}"
    }
}

fun main() {
    apply(listOf(1,2,3)) {
        if (it % 2 == 0) {
            println("偶數")
            return // <-- 報錯! 'return' is not allowed here
        }
        println("奇數")
        return // <-- 報錯! 'return' is not allowed here
    }
}
```

遞迴轉迴圈

有些方法會以自己遞迴呼叫自己的方法撰寫 有些時候這種情況會造成效能上的消耗 kotlin 可以自動將這種方法轉換為迴圈呼叫的方式 開發者可以不用自己去轉換

修飾子：`tailrec`

```
tailrec fun factorial(n:Int, result:int = 1):Int {
    if (n == 1) {
        return result
    }
    return factorial(n-1, result*n)
}
```

並不是所有遞迴呼叫 kotlin 都能自動轉換成迴圈呼叫，需要滿足以下條件

- 方法最後一個宣告 若且唯若 是呼叫該方法
- 上述宣告除了呼叫方法以外不能出現其他運算符號，即上述範例中最後一行如果是 `return factorial(n-1, result*n) * 1` 就不行，不行了話表示 `tailrec` 失敗，方法還是保持遞迴呼叫

集合內建 lambda 方法

map()

結構：`map(transform:(T)->R):List<R>`

將一個 T 型別集合內的元素帶入 transform 操作後回傳 R 型別的回傳值並把結果集合成 List

```
val numbers = intArrayOf(1,2,3,4,5)
val squares = numbers.map { it * it } // 1, 4, 9, 16, 25
```

flatMap()

結構：`flatMap(transform:(T)->Iterable<R>):List<R>`

將一個以上 T 型別集合中的元素一起做 transform 操作最終集合成一個 R 型別的 List

```
val numbers1 = listOf(1,2,3,4,5,6)
val numbers2 = listOf(10,20,30)

listOf(numbers1, numbers2).flatMap { it + 1 } // 2,3,4,5,6,7,11,
```

如果只是要把多個同型別集合平拍在一起，沒有要另外操作，可以使用 `flatten`

```
listOf(numbers1,numbers2).flatten() // 1,2,3,4,5,6,10,20,30
```

zip()

結構：`zip(other:Array<out R>):List<Pair<T,R>>`

連接兩個集合中的數據 以一對一的方式進行組合成 Pair 並將 Pair 物件集合成 List 最終結果會取決於長度較短的那個集合

```
val numbers1 = listOf(1,2,3,4,5,6)
val numbers2 = listOf(10,20,30)
println(numbers1.zip(numbers2)) // [(1, 10), (2, 20), (3, 30)]
```

reduce()

結構：`reduce(operation:(acc:S,T)->S):S`

對集合中元素做累計操作 累計初始元素為集合中第一個元素 每一次將計算結果再次帶入 `acc` 參數並以此跌代值到最後一個元素

```
listOf(1,2,3,4,5,6).reduce { acc, n -> acc - n }
//初始 acc=1, n=2 並將 1-2 的結果帶到下一輪的 acc
// 1-2-3-4-5-6 = -19

listOf(1,2,3,4,5,6).reduce { acc, n -> acc + n } // 1+2+3+4+5+6=
```

filter()

結構：`filter(predicate:(T)->Boolean)List<T>`

對集合中元素做過濾，最終集合只會有經過 `predicate` 運算後回傳 `true` 的元素

```
listOf(1,2,3,4,5).filter { it % 2 == 0 } // [2,4]
```

forEach()

結構：`forEach(action:(T)->Unit):Unit`

對集合元素進行輪巡操作

```
listOf(1,2,3,4,5).forEach { print("$it->") } //1->2->3->4->5->
```

partition()

結構：`partition(predicate:(T)->Boolean):Pair<List<T>,List<T>>`

將資料依照 `predicate` 回傳 `true` 和 `false` 分成兩群，並收集到一個 Pair 物件中

Pair 第一個元素表示 `predicate` 回傳 `true` 的元素集合 List Pair 第一個元素表示 `predicate` 回傳 `false` 的元素集合 List

```
val c = listOf(1,2,3,4,5).partition { it % 2 == 0 } // ([2, 4],
```

全是 RuntimeException

Kotlin 沿用 Java 例外處理機制，所有例外都是 Throwable 的子類別 和 Java 不同的是

Kotlin 沒有強制例外檢查機制，所有的例外都是 RuntimeException

開發者需要自行決定是否要對例外進行捕捉

例外捕捉

`try-catch-finally` 區塊

和 Java 一模一樣

try 表達式

`try-catch-finally` 也可以當作表達式 需要注意的是，當作表達式的時候， `finally` 會執行，但是回傳值會忽略，不會賦值給變數

```
val err = try {
    1 / 0 // 除以 0
} catch(e:ArithmeticException) {
    e.printStackTrace()
    -1 // 回傳 -1
} finally {
    println("finally execute")
    -2 // 回傳 -2
}

println("err=$err")

/**
java.lang.ArithmeticException: / by zero
    at MainKt.main(Main.kt:193)
    at MainKt.main(Main.kt)
execute finally // finally 還是會執行
err = -1 // finally 回傳的 -2 被忽略，被賦予 catch 中的 -1
*/
```

如果沒有 catch 到的 Exception， `finally` 區塊還是會執行的，不過會在賦值的地方噴出例外，就不會往下執行

```
val err = try {
     1 / 0
} finally {
     println("execute finally")
 -2
}

println("err = $err")

/**
execute finally
Exception in thread "main" java.lang.ArithmeticException: / by z
    at MainKt.main(Main.kt:193)
    at MainKt.main(Main.kt)
*/
```

DSL

先來談談 DSL 是什麼

Domain Specific Language

> 領域 指定 語言

DSL 意思是指 只用於解決某些特定領域問題 的語言

例如 SQL 就是一種 DSL，只能用於解決資料庫這一個領域的語言，開發者基本上無法使用它來撰寫一個完整的應用程式

而我們常常上手的那些程式語言如 Java、Kotlin、Golang、JS 則稱為 **通用程式語言 (General Purpose Programming Language)**

DSL 有兩個明顯的特點：

- 比通用語言省略大量的程式碼，能夠更簡潔的表達在該領域中執行的各種操作
- 通常更符合人類語言習慣

常見 DSL

- SQL : 資料庫操作
- Regex : 正則表達式
- Gradle、SBT : 建構工具
- Freemarker、JSX、Anko : 前端頁面生成

DSL 分類

- 外部 DSL : 保存在應用程式之外，一般應用很難直接和外部 DSL 進行互相呼叫

> 像 SQL，Java 就無法直接呼叫，得透過 JDBC 這類的實現 TCP 通訊底層的函式庫 (驅動程式)
>
> - 內部 DSL : 在應用程式中透過通用程式語言實現 DSL 語言，簡單來說就是 用語言來創造語言， 內部 DSL 通常和實際應用程式採用的通用程式語言進行撰寫，所以可以很容易和應用程式互動

Kotlin DSL

在 Kotlin 中實現 DSL 假設一個 Kotlin 的 Class

```
data class CPU(var core:Int, var arch:String)
```

要創建這個 Class 的物件基本方法就是

```
val cpu = CPU(8, "64 bit")
```

1. 使用 Lambda 表達式來實現 DSL

1. 先調整一下 `CPU` Class

```
class CPU(var core:Int = 1, var arch:String = "32 bit") {
    fun core(core:Int) {
        this.core = core
    }

    fun arch(arch:String) {
        this.arch = arch
    }
}
```

1. 創建一個叫做 `cpu` 的方法，參數為接受 CPU 參數的匿名方法，回傳 CPU 物件

```
fun cpu(block:(CPU)->Unit):CPU {
    val cpu = CPU() // 創建 CPU 物件
    block(cpu) // 把此物件當作參數呼叫匿名方法
    return cpu // 回傳新建的 CPU 物件
}
```

1. kotlin 中方法最後一個參數如果是匿名方法，則可以在呼叫的時候把 `{}` 放到 `()` 之後

```
// cpu({}) -> cpu() {} -> cpu {}
val c1 = cpu { c -> // 匿名方法架構是輸入一個 CPU 參數，沒有回傳
    c.core(2)
    c.arch("64 bit")
}
println("c1: core:${c1.core}, arch:${c1.arch}") // c1: core:2, a
```

1. Kotlin 中如果方法參數只有一個，會自動注入隱藏參數 `it` 裡面

```
val c2 = cpu {
    it.core(2)
    it.arch("64 bit")
}
println("c2: core:${c2.core}, arch:${c2.arch}") // c2: core:4, a
```

1. 如果再把 `cpu` 方法的參數改成傳送 CPU 類別的 **擴展方法**

```
fun cpu(init:CPU.()->Unit):CPU {
    // 參數為 CPU 類別的一個擴展方法，名稱是 init，此方法無輸入參數也
    val cpu = CPU()
    cpu.init()
    return cpu
}
```

因為是 CPU 類別的擴展方法，所以方法內可以直接存取到物件屬性 使用上就可以更簡化：

```
val c3 = cpu {
    // 整個方法內容是 CPU 類別的擴展，可以直接呼叫或使用 CPU 物件的方
    this.core = 8
    println("Oops I'm in closure") // 呼叫此擴展方法時就會 print
    arch("64 bit")

}
println("c3: core:${c3.core}, arch:${c3.arch}")

/**
Oops I'm in closure
c3: core:8, arch:64 bit
*/
```

這種用法在 Gradle 上非常常見

2. 使用 Fluent design 來實現 DSL

fluent design 是一種設計方式，透過方法回傳物件本身來達到連續呼叫方法的用法

1. 調整一下 CPU Class

```
class CPU(var core:Int = 0, var arch:String = "32 bit") {
    fun core(core:Int):CPU {
        this.core = core
        return this // 回傳自己
    }

    fun arch(arch:String):CPU {
        this.arch = arch
        return this // 回傳自己
    }
}
```

1. 使用上就可以像是 builder pattern 一樣做物件初始化

```
val cf = CPU().core(2).arch("64 bit")

println("cf: core:${cf.core}, arch:${cf.arch}") //cf: core:2, ar
```

1. 搭配 kotlin 方法中的 `infix` 修飾子，讓 `core` 和 `arch` 方法變成一種操作子 (Operator)

```
class CPU(var core:Int = 0, var arch:String = "32 bit") {
    infix fun core(core:Int):CPU {
        this.core = core
        return this // 回傳自己
    }

    infix fun arch(arch:String):CPU {
        this.arch = arch
        return this // 回傳自己
    }
}

val cf2 = CPU() core 4 arch "64 bit"

println("cf2: core:${cf2.core}, arch:${cf2.arch}") // cf2: core:
```

比較

Lambda 實現 DSL

```
cpu {
    this.core = 4
    arch("64 bit")
}
```

fluent design 實現 DSL

```
CPU() core 4 arch "64 bit"
```

模仿 Gradle 的 dependencies 區塊

```
class MyGradle {
    private val libs = mutableListOf<String>()
    fun implementation(libPath:String) {
        libs.add(libPath);
    }
    fun build() {
        println(libs)
    }
}

fun myDependencies(init:MyGradle.()->Unit) {
    val myGradle = MyGradle()
    myGradle.init()
    myGradle.build()
}
```

使用上就可以這樣用

```
myDependencies {
     implementation("org.jetbrains.kotlin:ohohohoho-whatever:0.0
    implementation("org.jetbrains.kotlin:ohohohoho-whatever:0.0.
    implementation("org.jetbrains.kotlin:ohohohoho-whatever:0.0.
}
```

執行會創建 `MyGradle` 物件並且執行完 `init()` 後會呼叫 `build()` 方法，印出

```
[org.jetbrains.kotlin:ohohohoho-whatever:0.0.1, org.jetbrains.ko
```