函式

- **簡短**

準則一，要簡短。

準則二，比準則一更短。

if、else、while ...（以及其他關鍵字）裡應只有一行敘述，那行敘述通常都是一個命名不錯的函式，可以使得閱讀程式像是在看故事一樣輕鬆。

縮排最好別超過一或兩層，在助於理解的前提下封裝演算法有其必要~~你不要覺得麻煩~~。

ex:

```
if(isSick(kb)){
    goToHospital(kb);
}
```

『是否生病』
『送去醫院』

- **只做一件事情**

判斷依據為『從函式命名所揭露的抽象概念與函式執行的實際行為是否一致』，或是『能否從原函式提煉出新的函式（包含新的抽象概念）』~~WTF~~。

ex:

```
public void meetingAndExecuteTask(Task task) {
    meeting();
    execute(task);
}
```

此函式從命名揭露一個總括的概念『開會以及執行任務』，實際行為上也一致，所以這是**只做一件事情**。

- **要無副作用**

『過平交道』停看聽這是一個承諾，結果 KB 『過平交道』除了停看聽還綁了鞋帶。

KB 並沒有做到只做一件事情，還有可能因為綁了鞋帶~~柵欄突然放下來嚇死~~影響其它路人。程式面來說，如果一個函式偷偷的做了某件事情而無法從函式命名中輕易聯想，就會造成維護上的困擾。

- **每個函式只有一層抽象概念**

上圖的函式『開會以及執行任務』是一個同一層次抽象概念的範例。

開會應包含鞠躬敬禮主席致辭一些實作細節，但不在此函式層次中揭露（同執行任務）。

函式表達的概念與實作細節若不在同一層次，會造成讀者不爽困惑以及維護上的困擾。

- **使用具備描述能力的名稱**

Ward Cunningham 說：當每個你看到的程式，執行結果都與你想的差不多，你會察覺到你正工作在 clean code 之上。(註1)

別害怕排斥花時間更換更精準的函式名稱。

現在的 IDE 很強，請把函式名稱更換到你認為描述能力夠強為止。

在專案裡請統一使用相同的命名慣例，最後會發現函式執行結果 … 都和你想的差不多。

- **動詞和關鍵字**

除了可以被搜尋的原則外，函式該如何命名？需要良好的解釋函式的意圖或參數順序性。

1. 舉例單一參數的情況下可以使用『動詞（名詞）』或是『動詞+參數本質（名詞）』
   - execute(task) 使用了『動詞（名詞）』
   - executeProject(task) 使用了『動詞＋參數本質（名詞）』，更清楚 task 是專案中的任務。
2. 舉例雙參數函式表達參數順序性 ex: checkNameAndPassword(name,password)

- **函式的參數**

最理想是零參數，最多兩個參數。

超過三個參數除非有特殊理由，無論如何都不該如此。

參數有透露概念的能力，當參數越多讀者就更需要去了解相依關係以及追蹤來源導致思緒中斷，也容易造成單元測試上的困難。

- **旗標參數（Flag）**

這代表著相依函式可能不止做一件事情，這是~~非常爛~~不太妥當的做法。

應合理的將原函式分裂成兩個函式，函式應**只做一件事情**。

- **單一參數的常見形式**

常見的三種情境如下 …

1. 與這個參數有關的問題。

    ex: boolean isSick(customer)

2. 對這個參數進行某種操作，將該參數轉換成某種東西然後**回傳**。

    ex: Customer getCustomer(id)

3. 某種事件修改狀態的無回傳值函式。

    ex: void updateValidateFailCount(customer)

敘述 3 是一種**事件**。改變系統資訊的**事件**，請謹慎命名以及給予良好的命名空間。

若不是上述的狀況，請試著別使用單一參數形式函式。

- **兩個參數的函式**

無法避免，但請有效的表達參數順序性以及**使用具備描述能力的名稱**。

如果允許，請合理的合併物件使函式更加整潔且降低參數與函式間的耦合~~這並不算作弊~~。

ex:

```
void execute(Task task,People people);
void execute(Task task);
```

『執行任務』函式將 People 物件合併至 Task 物件。

- **三個參數的函式**

讀者容易有參數順序性、追蹤參數來源中斷原演算法思考以及單元測試的困難，使用三參數以上函式前需謹慎考量是否合理以及必要~~最好燒毀~~。

- **使用例外處理取代回傳錯誤代碼**

ex:

```
if (Status.OK == recordCustomerBeforeDelete(customer)) {
    // do something ...
    if (Status.OK == deleteCustomer(customer)) {
        // do something ...
    } else {
        // do error control ...
    }
} else {
    // do error control ...
}
```

- 使用代碼
呼叫者必須處理非預期的代碼，將導致更深層的巢狀結構。

```
try{
    recordCustomerBeforeDelete(customer);
    deleteCustomer(customer);
}catch(Exception e){
    // do error control ...
}
```

- 使用例外處理
統一處理錯誤資訊，使程式更為整潔。

- **提取 Try / Catch 區塊**

若 try / catch 混在程式中，讀者容易混淆程式的結構。

將 try / catch 往外提取，使得原函式更加清楚。

ex:

```
public void delete(Customer customer) {
    try {
        deleteCustomerReferences(customer);
    } catch (Exception e) {
        // do error control ...
    }
}
```

- **錯誤處理就是一件事情**
所以在 catch / finally 之後除了回傳值理應不會有其他任何函式敘述句。

```
protected void deleteCustomerReferences(Customer customer) {
    recordCustomerBeforeDelete(customer);
    deleteCustomer(customer);
}
```

deleteCustomerReferences 將原演算法封裝成一個層次較高的抽象概念實作。

- **Error.java 的依附性磁鐵**

假設使用常數檔或列舉檔去定義錯誤的代碼或是列舉，當有新的錯誤定義時我們就必須去更改這些檔案。

但如果是使用例外處理，僅需新增錯誤定義的衍生類別。 (註2)

- **指令和查詢的分離**

函式要能解決某種問題（query）或是做某件事情（command），但是不應該同時發生。

ex:

```
if (setCustomer(customer)) {
    // do something
}
```

setCustomer 可能會讓讀者混淆，語意上是設定但是卻寫在 if block

```
if (customerExist(customer)) {
    setCustomerAttribute(customer);
    // do something
}
```

customerExist 判斷是否存在
setCustomerAttribute 針對 customer 物件操作

- **Switch 敘述**

函式內的 switch 敘述，如果是用來產生多型物件，則應使用工廠模式 (註3) 將 switch 程式片段埋在工廠抽象函式實作底下。

當有新的多型物件衍生類別僅需要更改工廠抽象函式實作，將不影響到原演算法。

- **不要重複自己**

所有軟體相關產業最邪惡的根源。

一段程式碼重複 N 次於應用程式中，當該段程式邏輯有異動就必需要改 N 次~~9487995~~。

註

1. **Ward Cunningham**：計算機程式設計師和Wiki概念的發明者。他也是設計模式和敏捷軟體方法的先驅之一。
2. **開放封閉原則**：類別應該開放，以便擴充；應該關閉，禁止修改。
3. **工廠模式**：定義了一個建立物件的介面，但由次類別決定要實體化的類別為何。工廠方法讓類別把實體化的動作，交由次類別決定。