前言

本書背後的哲學

資料結構和演算法在過去五十年的重大發明中隨處可見，而且是軟體工程師必備的知識基礎。但以我個人的觀察來說，這類主題書都太理論、太厚、太“硬”了：

太理論

演算法的數學分析會簡化假設，往往導致限制實務上的用途。很多這類主題只專注在數學卻忽略了那些假設。所以在這本書中，我將以最實務的重點來介紹演算法。

太厚

大多數這類書籍都超過五百頁，有些甚至高達一仟頁。我將關注對軟體工程師最有用的主題，將這本書的頁數壓在一百五十頁以內。

太“硬”

很多資料結構的書籍都著重在如何實作資料結構，很少講述如何使用它們。在這本書中，我會自介面開始“從上到下”進行介紹。讀者將從學習如何使用 Java Collections Framework 中的資料結構開始，然後才進入到內部運作。

最後，有些書就是一直不停介紹不同的資料結構而不談為何要用它們，讀起來實在痛苦！我試著用網頁搜尋這個的實際應用來貫穿許多資料結構討論，網頁搜尋是一種重要的應用，而且應該也比較讓人覺得有趣。

網頁搜尋會用到一般資料結構課程不會說到的一些主題，例如使用 Redis 保存資料。

決定一些內容的去留對我來說並不容易，但最後我也作了一些妥協。我引入了一些題目，它們是多數讀者不會用到，但在面試時會被問到的主題，針對這些主題，我會說明一般認知為何，外加我自己主觀看法。

本書也有軟體工程實務面的主題，包括版本控制和單元測試。多數的章節會有章節練習，讓讀者練習一下剛學習到的東西，每個練習都會有自動驗證來檢驗正確性，我也會把我版本的解答放在下一章的開頭。

讀者背景

本書是設計給大學計算機系或相關科系學生、專業軟體工程師、學習軟體工程者或是準備應試科技工作者而準備。

在你開始這本書之前，你應該已經對 Java 有一定的熟悉程度，特別是你應該知道如何定義繼承新類別、或是實作新的 `interface`，如果你對 Java 不熟悉，可以參考閱讀以下兩本書：

- Downey 和 Mayfield 所著的《*Think Java*》（O'Reilly Media, 2016），適合未有寫程式經驗的人閱讀。
- Sierra 和 Bates 所著的《*Head First Java*》（O'Reilly Media, 2005），適合在 Java 之外，已有其他程式語言經驗的人閱讀。

如果你對 Java 的 interface 不熟悉，可以在 *http://thinkdast.com/interface* 上閱讀一篇名叫“What Is an Interface? ”的導覽。

“interface”這個字很容易令人混淆，從 **application programming interface**（API）方面來看，它的意思是提供特定功能的一群類別和方法集合。

然而在 Java 的世界裡，它則是一種程式語言功能，概念類似類別，是一群特定方法的集合。為了要避免混淆，我將會使用“介面”兩字來代表一般認知的介面，用 `interface` 來表示 Java 語言功能中的 `interface`。

讀者應已瞭解型態參數（type parameter）和一般型態的差別，舉例來說，你應該知道如何利用型態參數來建立一個新物件，例如 `ArrayList<Integer>`，如果不知道的話，可以在 *http://thinkdast.com/types* 讀到相關的資訊。

讀者應該已經對 Java Collections Framework（*http://thinkdast.com/collections*）感到熟悉，特別是 `List` interface、`ArrayList` 和 `LinkedList` 類別。

若讀者能熟悉 Apache Ant 更佳，它是一種 Java 的自動建置工具，你可以在 *http://thinkdast.com/anttut* 取得更多資訊。

最後，讀者應已瞭解 `JUnit`，它是 Java 用來作單元測試的 framework，你可以在 *http://thinkdast.com/junit* 取得更多說明。

本書資源

這本書的程式碼都可以在 Git repository 取得，位置是 *http://thinkdast.com/repo*。

Git 是一種**版本控制系統**，讓你可以持續追蹤專案裡檔案的改變，而 **repository** 是指 Git 裡的檔案集合。

GitHub 是提供儲存 Git repository 的一個伺服器，它有著方便的網頁介面，有以下數種功能：

- 你可以按下 Fork 按鈕，來複製 GitHub 上的一個 repository，如果你還沒有 GitHub 帳號，你需申請一個。執行完 fork 之後，你就會在 Github 上擁有一份自己的 repository，可開始用來追蹤自有版本程式碼的改變，之後可以 **clone** 一個 repository，clone 讓你下載一份程式碼檔案到你的電腦中。
- 也可以只作 clone 不作 fork，如果你選擇這麼做的話，你就不需要 GitHub 帳號，不過你也無法利用 GitHub 儲存改變。
- 如果你完全不想使用 Git，你可以在 GitHub 頁面上按下 Download 按鈕，或是從連結 *http://thinkdast.com/zip*，用 ZIP 壓縮方法下載程式碼。

當你從 repository clone 到本地端或解壓 ZIP 取得程式碼後，你應該會看見一個叫 `ThinkDataStructures` 的目錄，下面有個叫 `code` 的子目錄。

本書的範例使用 Java SE Development Kit 7 開發和測試，如果你使用的是舊的版本 ，可能導致部份範例無法執行，如果是用更新的版本，那使用上應該就不會有問題。

第一章

介面

本書有三個主題：

資料結構（*Data structure*）

從 Java Collections Framework（JCF）中的資料結構開始，你會學習到如何使用像是 list 或 map 這種結構。

演算法分析（*Analysis of algorithms*）

將說明分析程式碼的技巧，預測程式可以跑多快，以及會使用到多少空間（記憶體）。

資料檢索（*Information retrieval*）

為使前面兩點以及練習題更加有趣，我將使用資料結構和演算法來製作一個網頁搜尋引擎。

這三個主題，在本書中將以下面的大綱結構呈現：

- 從 `List` 開始，你會為 `List` interface 實作兩個類別，然後你可將自己的實作與 Java 類別中的 `ArrayList` 和 `LinkedList` 作比較。
- 接著我們介紹樹形的資料結構，此時你會進行第一個應用實作，也就是寫一個程式讀取 Wikipedia 網頁的內容，拆解內容以及在拆解出的結果樹裡找到連結。我們會用它來測試“找到 Philosophy”這個推測（你可以在 *http://thinkdast.com/getphil* 找到說明）。
- 然後我們會學習 `Map` interface 以及 Java 中的 `HaspMap` 實作，接著用雜湊表（hash table）和二元樹來實作這個介面。

- 最後，你會使用以上的類別（還有一些我過程中會介紹的類別）實作一個網頁搜尋引擎，包括找尋和讀取頁面的網路爬蟲（crawler），能用特定格式儲存網頁內容，使得搜尋加快的索引器（indexer），還有一個接收使用者查詢並返回結果的檢索介面。

那麼，就讓我們開始吧。

為什麼需要兩種 List

剛開始接觸 Java Collections Framework 的人，面對 `ArrayList` 和 `LinkedList` 都會產生一個問題，也就是為什麼 Java 要同時提供兩種 `List` interface 的實作？以及如何選擇使用哪一種？我將會在接下來的幾章陸續回答這些問題。

我將會從查看 `interface` 和它的實作類別開始，然後說明什麼是“介面程式設計”（programming to an interface）。

最前面幾個練習是會請你實作很相似於 `ArrayList` 與 `LinkedList` 的類別，目的是要讓你知道它們的工作原理，然後我們也會說明兩者的優缺點。有些工作選用 `ArrayList`，能用較少空間或較快速度完成，而有些工作則是用 `LinkedList` 更適合。什麼情況適用哪種，要視應用中最吃重的工作來決定。

Java 中的 Interface

一個 Java 的 `interface` 就是一個指定方法的集合，任何想實作特定 `interface` 的類別，就要具備滿足 `interface` 的指定方法。舉例來說，下面是一個 `java.lang` package 中的 `Comparable interface` 的程式碼。

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

這個 `interface` 的定義用了一個型態參數 `T`，它讓 `Comparable` 變成**泛型**（**generic type**），要實作這個 `interface` 的類別必須要：

- 指定 `T` 是什麼
- 實作一個叫 `compareTo` 的方法，這個方法有一個物件當作參數，並回傳 `int`。

以 java.lang.Integer 程式碼來當例子：

```
public final class Integer extends Number implements Comparable<Integer> {

    public int compareTo(Integer anotherInteger) {
        int thisVal = this.value;
        int anotherVal = anotherInteger.value;
        return (thisVal<anotherVal ? -1 : (thisVal==anotherVal ? 0 : 1));
    }

    // 省略其他方法
}
```

這個類別繼承了 Number，所以它擁有 Number 的所有方法和變數，並且它指定實作 Comparable<Integer>，所以必須實作 compareTo 方法，它所實作的 compareTo 方法有一個 Integer 參數，並回傳 int。

當一個類別宣告要實作一個 interface 時，編譯器會檢查它是否已具備該 interface 所定義的所有方法。

順帶一提，在 compareTo 實作中使用了 ?:，它是「三元運算子」（ternary operator），如果你對這個運算子的用法不熟悉，可以參考 *http://thinkdast.com/ternary*。

List interface

Java Collections Framework（JCF）中定義了一個叫 List 的 interface，它有 ArrayList 和 LinkedList 兩種實作。

List interface 是用來定義 List 必須具備的條件，想實作這個 interface 的任何類別，都必須提供特定方法的集合，包括 add、get、remove 和其他大概 20 個方法。

ArrayList 和 LinkedList 也都具備了指定的方法，所以它們可以相互替換使用，也就是說一個會使用到 List 的方法，也可以用 ArrayList、LinkedList 或其他符合 List interface 的類別取代。

讓我們用一個例子來說明：

```
public class ListClientExample {
    private List list;

    public ListClientExample() {
        list = new LinkedList();
```

```
    }

    private List getList() {
        return list;
    }

    public static void main(String[] args) {
        ListClientExample lce = new ListClientExample();
        List list = lce.getList();
        System.out.println(list);
    }
}
```

ListClientExample 類別沒有做什麼事，但這個類別**封裝**了 List，也就是它有一個 List 的變數實作，我用這個類別說明概念，而你將在第一個練習題中再度使用到這個類別。

ListClientExample 建構子中，建立一個新的 LinkedList，並將 list 作了**實例化**（也就是建立 list）。用來取得 list 的方法叫 getList，它會回傳內部 List 物件的參照，而 main 中有幾行程式碼用來呼叫這個方法。

在這個範例中的重點，是它盡可能地使用 List，若非必要不去特別指定是 LinkedList 或 ArrayList，舉例來說，初始變數被宣告成 List，getList 方法也是回傳 List，兩者都沒有指定實際上用的是哪一種 list。

如果你中途變心，想要改用 ArrayList 的話，只要改變建構子的內容，其他的程式都不需要修改。

這個寫作的方法稱為**介面程式設計**（**interface-based programming**），或是稱為“programing to an interface”（請見 *http://thinkdast.com/interbaseprog*），這裡的介面指的是一般定義的介面，而不是 Java 的 interface。

當你使用函式庫時，你的程式碼應要盡量使用像是 List 的這種 interface，而不是指定使用像 ArrayList 這樣的實作。這樣一來，未來如果要改動的話，程式碼也比較好改。

另一個方面來說，如果介面變更的話，相關的程式碼也要跟著一起改。所以若不是必要的話，函式庫開發者都會避免改動介面。

練習題一

由於這是第一個練習，所以不會太複雜，就請你從前面的程式碼中，將實作替換，也就是用 ArrayList 替換掉 LinkedList。由於程式碼本身就是基於介面實作的，所以應該只要改寫一行並加上 import 述句即可。

要做這個練習之前，首先你要設定開發環境，之後所有的練習題，都需要編譯及執行 Java 程式碼。我使用 Jave SE Development Kit 7 來開發範例程式，如果你用的是更新的版本，應該不會碰到問題，不過如果你用的是舊的版本，有可能會有一些相容性問題發生。

我建議使用整合開發環境（Integrated development environment, IDE），這種開發環境提供了語法檢查、自動編譯和原始碼整理的功能。這些功能幫你避免及快速找到錯誤。不過，如果你的目的是要準備面試的話，面試時可不會有這些工具，所以基於這個考量，你也可以不使用這種工具來進行程式碼的撰寫。

如果你還沒有下載本書的程式碼的話，可以看第 ix 頁“本書資源”裡的說明。

在本書的程式碼中，有一個叫 code 的目錄下，你可以找到以下的檔案和目錄：

- build.xml 是一個 Ant 檔，用來幫助編譯和執行程式碼。
- lib 目錄下包含你會需要的所有函式庫（在這個練習中，只用到 JUnit）。
- src 目錄下包含了所有原始碼。

在 src/com/allendowney/thinkdast 下，你可以找到這個練習題的程式碼：

- ListClientExample.java 是前一節用的範列程式。
- ListClientExampleTest.java，是一個包含了 JUnit 測試的 ListClientExample。

重讀一次 ListClientExample 並確定你瞭解它在幹嘛，然後編譯並執行它，如果你使用 Ant 的話，你可以到 code 目錄下執行 ant ListClientExample。

此時可能會出現警告訊息：

```
List is a raw type. References to generic type List<E>
should be parameterized.
```

為了保持練習題的單純性，範例中並未指定 List 使用元素的型態，如果這讓你很困擾的話，可以將 List 或 LinkedList 改為 List<Integer> 或 LinkedList<Integer>。

查看一下用來測試的 ListClientExampleTest，它執行後會建立 ListClientExample，呼叫 getList，然後檢查回傳型態是不是 ArrayList。由於原本回傳型態是 LinkedList 而不是 ArrayList，所以這個檢查預設一定會失敗，請你執行這個測試，並確認結果為失敗。

注意：這個測試雖然是給練習題使用，但它不是一個好的測試範例，好的測試程式應該是去檢查受測類別是否符合 *interface* 規範，而不是檢查實作是什麼型態。

請在 ListClientExample 中，用 ArrayList 取代 LinkedList，也許你已加好了 import 述句，那麼就只要編譯執行 ListClientExample 即可，做完後，再執行一次測試，改造過程式碼應該就會測試通過了。

只要把建構子中的 LinkedList 改掉就可以讓測試成功了，不要改動其他 List 出現的地方，不過如果你改了的話會怎樣呢？試試看把幾個 List 出現的地方改為 ArrayList，結果程式碼還是可以執行，不過這樣一來程式碼就會變得「僵化」，若你未來想要改變 interface 的話，你就要作多處程式碼改動。

在 ListClientExample 建構子中，若你把 ArrayList 取代成 List 會怎樣呢？你覺得 List 不能被實例化是為什麼呢？

第二章

演算法分析

如前一章看到的，Java 提供兩種 List interface 的實作，也就是 ArrayList 和 LinkedList，有一些應用配合 LinkedList 速度較快，而另一些則是要配合 ArrayList 才會比較快。

想知道特定的應用程式適合使用哪種實作的話，最土法煉鋼的方法就是兩個都跑跑看，然後看看各花了多少時間，這個方法稱為**性能分析**（**profiling**），它有幾個問題存在：

1. 你必須先兩種都實作才能進行分析。
2. 分析的結果可能和電腦種類相關，一種演算法適合在某台機器上跑，但另外一種可能適合另外一台。
3. 這個分析結果可能受到規模或輸入資料的影響。

我們也可以用**演算法分析**（**analysis of algorithms**）來知道答案，演算法分析可以在不實作的情況下，藉由比較演算法的方法得到結果，不過這方法有一些前提假設：

1. 為了排除電腦硬體造成的影響，通常會找出要用到的基礎運算，例如加、乘和數字比較等，計算它們在一個演算法裡要用到的次數。
2. 為避免輸入資料的影響，就是把所有輸入值執行結果作平均，如果這點無法做到，通常就是只取最差狀況結果。
3. 最後，還要處理演算法對於小問題表現優良，但大問題就很糟的情況。這種情況我們通常就把焦點放在大問題上，因為小問題就算執行結果有差異，在總時間花費上可能差異也不大，但是大問題的結果差異可能會很巨大。

這類分析方法在演算法裡自成一格，舉例來說，若我們知道演算法 A 的執行時間和輸入個數 n 呈相關，而演算法 B 與 n^2 成相關，那麼就可以說演算法 A 比 B 快，特別是 n 值越大時越明顯。

大多數的演算法執行時間可以用以下分類：

常數時間（*Constant time*）

一個演算法為**常數執行時間**，表示執行時間與輸入數量無關，舉例來說，如果你有一個具有 n 個元素的陣列，然後你用中括號（`[]`）取得元素值，這個動作不管陣列多大，執行時間都是一樣的。

線性時間（*Linear*）

一個演算法為**線性執行時間**，表示與輸入的數量呈相關，舉例來說，如果你想把陣列元素加總，那你就得進行存取 n 個元素，並執行 $n-1$ 次加法，動作的總數是 $2n-1$（存取的元素數量和加法次數），也就是和 n 相關。

平方時間（*Quadratic*）

一個演算法若是**平方執行時間**，表示它與 n^2 相關，舉例來說，如果你想要檢查一個 list 中的元素是否有出現超過一次，一個簡單的演算法就是將每個元素都和其他元素作比較，如果總數是 n 個元素，每個又執行 $n-1$ 次比較，那麼動作的總數就是 n^2-n，也就是和 n^2 呈相關。

選擇排序法

舉例來說，下面的程式是一個稱為**選擇排序法**（**selection sort**）的簡單演算法（參考 *http://thinkdast.com/selectsort*）：

```
public class SelectionSort {

    /**
     * 將索引 i 和 j 的元素互換
     */
    public static void swapElements(int[] array, int i, int j) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
    /**
     * 從 start 處找到最小數值的索引值
```

```
     * 一直到陣列結束
     */
    public static int indexLowest(int[] array, int start) {
        int lowIndex = start;
        for (int i = start; i < array.length; i++) {
            if (array[i] < array[lowIndex]) {
                lowIndex = i;
            }
        }
        return lowIndex;
    }

    /**
     * 用選擇排序法將陣列中的值重排
     */
    public static void selectionSort(int[] array) {
        for (int i = 0; i < array.length; i++) {
            int j = indexLowest(array, i);
            swapElements(array, i, j);
        }
    }
}
```

第一個方法 `swapElements`，用來交換陣列的兩個元素，由於我們已知元素數量和陣列啟始位置，讀取和寫出元素是靠一個乘法和一個加法執行，所以讀取和寫出是常數時間。又因為 `swapElements` 裡的東西都是常數時間可完成，所以歸納整個方法也是常數時間。

第二個方法 `indexLowest`，用來找從陣列指定啟始位置 `start` 開始找最小元素，迴圈每次執行都從陣列中取兩個元素作比較，由於這些都是常數時間動作，所以要計算哪個動作都一樣，為方便我們就計算比較這個動作：

1. 如果 `start` 為 0，`indexLowest` 會遍歷整個陣列，所以比較的次數就是陣列的長度，我們稱它 n。
2. 如果 `start` 為 1，那麼比較次數為是 $n-1$。
3. 一般來說，比較次數可以寫成 n – `start`，所以 `indexLowest` 是線性時間。

第三個方法 `selectionSort`，用來排序陣列。迴圈從 0 執行到 $n-1$，所以迴圈執行 n 次，每一次都會呼叫 `indexLowest`，並執行常數時間的 `swapElements`。

`indexLowest` 第一次被呼叫時，它執行 n 次比較，第二次被呼叫時，它執行 $n-1$ 次比較，以此類推，得到比較次數總共為：

$$n+n-1+n-2+...+1+0$$

上面式子可以寫為 $n(n+1)/2$，也就是和 n^2 相關，這也表示推得 `selectionSort` 的執行時間是平方時間。

若把 `indexLowest` 想成一個巢式迴圈也可以得到一樣的結果，每次 `indexLowest` 動作都和 n 相關，一共呼叫它 n 次，所以總數是 n^2。

Big O

所有的常數執行時間，都可以被稱為 $O(1)$，所以可稱一個常數時間演算法屬於 $O(1)$，相同的，所有的線性執行時間演算法屬於 $O(n)$，所有平方時間演算法為 $O(n^2)$，這個區分演算法的方法稱為 **Big O 標示法**（**Big O notation**）。

注意：我另外作了一個 Big O 標記法的定義，可參考 *http://thinkdast.com/bigo*。

這個標記法在組合演算法時，是一個方便撰寫規則的方法，比方說，如果你先執行一個線性時間演算法，再執行一個常數時間演算法，那麼總執行時間還是線性的，以下式子中 ∈ 符號代表“屬於”：

若 $f \in O(n)$ 且 $g \in O(1)$，則 $f+g \in O(n)$。

如你執行的是兩個線性時間演算法，那加總後還是線性：

若 $f \in O(n)$ 且 $g \in O(n)$，則 $f+g \in O(n)$。

事實上，如果你執行線性時間演算法 k 次，只要 k 是常數，而且和 n 沒有任何關連性，那結果還是線性的：

若 $f \in O(n)$ 且 k 是常數，則 $kf \in O(n)$。

但如果你執行線性演算法 n 次，那結果就會是次方了：

若 $f \in O(n)$，則 $nf \in O(n^2)$。

一般來說，我們只關心 n 的最大指數，所以若總執行次數為 $2n+1$，那它屬於 $O(n)$，開頭的係數 2 還有後面的 1，對我們這種分析法都不重要。相同的，如果是 $n^2+100n+1000$，就屬於 $O(n^2)$，別因為數字大就嚇到了！

時間複雜度（**order of growth**）也代表同一個意思，時間複雜度相同的演算法就是屬於同一個 Big O 分類的演算法。舉例來說，所有的線性演算法都是同一個時間複雜度，因為它們的執行時間都是 $O(n)$。

在英文名稱中的“order”一字，指的是一群的意思，就像講“Order of the Knights of the Round Table”（圓桌武士），意思是一群武士（騎士），而不是把這群人依序排排站的意思。所以你可以把“Order of Linear Algorithms”（線性時間演算法），想像成一群勇敢、聰明又特別有效率的演算法。

練習題二

本章的練習題是要實作用 Java 陣列作為儲存體的 `List`。

在本書隨附程式碼中（見第 ix 頁的“本書資源”），你可以找到需要的程式碼：

- `MyArrayList.java` 包含了一部份 `List` interface 要實作的程式碼，其中有四個方法是不完整的，你的練習就是要完成它們。
- `MyArrayListTest.Java` 包含了 JUnit 測試，你可以用來檢查答案是否正確。

你會在 `code` 目錄裡找到 Ant 用的 `build.xml` 檔，可以用 `ant MyArrayList` 命令來執行 `MyArrayList.java`，裡面內含了幾個簡單的測試，或是也可以用 `ant MyArrayListTest` 來執行 JUnit 測試。

當你執行測試時，可能出現數個失敗，如果你檢查程式碼的話，會看到四個 `TODO` 註解，用來說明要請你完成程式碼。

在你開始實作那些方法之前，讓我們先來看一些程式碼，下面的程式是類別定義、變數實例和建構子：

```
public class MyArrayList<E> implements List<E> {
    int size;                    // 記錄元素個數
    private E[] array;           // 儲存元素

    public MyArrayList() {
        array = (E[]) new Object[10];
```

```
            size = 0;
        }
    }
```

如註解中標明，size 是用來記錄 MyArrayList 中有多少元素，而 array 是實際拿來儲存元素的陣列。

建構子會建立一個包含 10 個元素的陣列，預設值為 null，並設定 size 為 0，在大多數時間，陣列的長度都會比 size 大，所以陣列裡總是會有空位。

提示一個 Java 的細節，就是無法用型態參數來初始化陣列，舉例來說，以下的程式碼是不合法的：

```
array = new E[10];
```

要避過這個問題，你就要先生出一個 Object 陣列，然後再對它作強制轉型，你可以在 *http://thinkdast.com/generics* 讀到更多相關資訊。

接著我們要看添加元素到陣列的方法：

```
public boolean add(E element) {
    if (size >= array.length) {
        // 製作一個更大的陣列，並將原本內容複製過去
        E[] bigger = (E[]) new Object[array.length * 2];
        System.arraycopy(array, 0, bigger, 0, array.length);
        array = bigger;
    }
    array[size] = element;
    size++;
    return true;
}
```

若陣列已沒有空位可用時，就會建立一個更大的陣列，並且複製原來的內容過去，接著就可以一樣將新元素儲存，並增大 size。

由於方法回傳值永遠都是 true，顯得這個回傳值沒有什麼意義，你可在 *http://thinkdast.com/colladd* 裡看到這麼做的理由。此外，這個方法的效能分析，也需要稍加討論一下，在一般的情況下，它是常數時間，但如果碰到要擴張陣列的情況，就會變成線性時間，我在會第 17 頁的“評估 add 方法”中作更清楚的解釋。

最後我們來看一下 get，看完以後你就可以開始著手進行練習了：

```
public T get(int index) {
    if (index < 0 || index >= size) {
```

```
            throw new IndexOutOfBoundsException();
        }
        return array[index];
    }
```

get 方法很簡單，如果索引值超界了，就丟出例外，否則的話就讀取並回傳陣列元素值。注意檢查超界是檢查 size，而不是檢查 array.length，所以也不會存取到未使用的陣列空位。

在 MyArrayList.java 中，有個長得像下方程式碼的 set：

```
    public T set(int index, T element) {
        // TODO: fill in this method.
        return null;
    }
```

閱讀 *http://thinkdast.com/listset* 上關於 set 的文件，並將方法內程式補齊之後，執行 MyArrayListTest，testSet 測試就會顯示結果成功了。

提示：試看看不要再寫一次檢查索引的程式碼。

你的下一個任務是實作 indexOf 的內容，和之前一樣，閱讀文件 *http://thinkdast.com/listindof* 後，你就知道自己該做什麼，這邊請特別留心對於 null 的處理。

我已提供一個叫 equals 方法，來協助從陣列裡找到目標值，並且在找到時回傳 true（已處理 null 值情況），注意該方法是 private，只能在這個類別中使用，它並不是 List 介面的一部份。

弄好了以後，現在執行 MyArrayListTest，testIndexOf 及它相依的測試應該也要能通過。

只剩 2 個方法你就完成這個練習，下一個是覆寫掉 add，讓它可以接受索引作為參數，並將新值依指定的索引值進行儲存，必要的話必須移動其他的元素以騰出空位。

一樣，先閱讀文件 *http://thinkdast.com/listadd*，實作並執行測試看看是否符合預期。

提示：請避免重複擴大列陣的程式碼。

最後一個，是要實作 remove，文件在 *http://thinkdast.com/listrem*，當你完成後，所有的測試應該都要通過。

都弄完了以後，你就有自己的實作版本了，請將它和我的版本比對一下， 我的程式放在 *http://thinkdast.com/myarraylist*。