侯捷觀點

Java 泛型技術之發展

— JDK1.4 上的實現

北京《程序員》2002.08
台北《Run!PC》2002.08

作者簡介：侯捷，臺灣電腦技術作家，著譯評兼擅。常著文章自娛，頗示己志。
侯捷網站：http://www.jjhou.com（繁體）
北京鏡站：http://jjhou.csdn.net（簡體）
永久郵箱：jjhou@jjhou.com

- 讀者基礎：有 Java 語言基礎，最好用過 Java Collection classes。
- 本文適用工具：(1) JDK1.4+JSR14 (2) Generic Java (GJ)。
- 本文程式源碼 (javag.bat, Test.java, Employee.java, JQueue.java)可至侯捷網站下載
- 本文同時也是 JavaTwo-2002 技術研討會之同名講題的書面整理與補充。

泛型技術細說從頭

泛型概念濫觴於 Doug McIlroy 於 1968 年發表的一篇著名論文 "*Mass Produced oftware Components*"，那篇論文提出了 "**reusable components**"（可復用軟體組件，又稱爲軟體積木或軟體 IC）的願景。過去數十年來，泛型技術比較屬於研究單位中的驕客，實作出來且被廣泛運用的產品極少。雖然 Ada, ALGOL68, Eiffel, C++ 等語言都支援泛型相關語法，但是直到 C++ STL 的出現，泛型技術在軟體發展圈內才開始有了大量迴響。

泛型（**generics, genericity**）又稱爲「參數化型別（**parameterized types**）」或模板（**templates**），或所謂「參數式的多型（**parametric polymorphism**）」。主要是一種型別代換（**type substitution**）概念，是和繼承（**inheritance**）不同而互補

的一種組件復用機制。泛型技術最直接被聯想到的用途之一就是建立資料群集（collections），允許使用者將某些特定型別的資料（物件）置入其中，並於取出時明確知道元素的型別，無需做（向下）轉型動作。假設某個泛型程式庫提供 list，你便可以明確宣告一個內含 int 或 double 或 Shape（使用者自定型別）元素的 list（這是 C++ 應用型式），或明確宣告一個內含 Integer 元素的 list（這是 Java 應用型式），如下：

```
list<int> iList; // in C++
LinkedList<Integer> iList = new LinkedList<Integer>(); // in Java
```

組件（此處狹義地指 classes）復用技術中，繼承和泛型的分野在哪裡？可以這麼說，**當你運用繼承**，面對不同的型別（types）你將擁有相同的介面，而那些型別獲得了多型性（多態性，Polymorphics）。**當你運用泛型**，你將擁有許多不同的型別，並得以相同的演算法（如排序、搜尋）作用在它們身上。舉個例子，發展繪圖系統時我們往往會設計一個 CShape，並令所有（幾何乃至非幾何）形狀皆衍生自它。爲了讓所有衍生自 CShape 的形狀都有自繪能力，並有相同介面 draw()，你必須使用繼承技術並令 draw() 爲一個虛擬函式。此處所以採用繼承和虛擬機制，是因爲 draw() 介面相同但演算法不同（不同的形狀當然有不同的繪圖法）：

```
// in C++
class CShape {
  public:
    virtual void draw()=0;
};
class CRect : public CShape { ... };     // CRect is a CShape
class CCircle : public CShape { ... };   // CCircle is a CShape
  // 每一個 CShape-derived classes 都必須覆寫 draw().
```

```
// in Java
public abstract class CShape {
  public abstract void draw();  // always virtual in Java
}
public class CRect extends CShape { ... }     // CRect is a CShape
public class CCircle extends CShape { ... }   // CCircle is a CShape
  // 每一個 CShape-derived classes 都必須覆寫 draw().
```

但是當我們欲設計一個「先進後出（FILO）」容器 Stack 時，情勢丕變。此時的情況是：演算法不因元素型態而有任何改變（只要「先進後出」即可。元素型別

並不會影響如何「先進後出」），我們應該採用泛型技術而非多型（繼承）技術：

```
// in C++ STL, <stack>
template <typename T> // T 爲未定（待定）型別
class Stack {
  T push(T elem);
  T pop();
  ...
};

Stack<int> iStack;
  // iStack 是個同質容器，每個元素都必須是 int，否則編譯器會報錯。
iStack.push(4);
iStack.push(6);
```

```
// in Java, jdk1.4\src\java\util\Stack.java
public class Stack extends Vector
    public Object push(Object item) { ... }
    public synchronized Object pop() { ... }
}
  // Java Stack 是個異質容器，每個元素的型別都是 Object。
  // 取出元素時由使用者自行向下轉型（downcast）爲正確型別 — 這使得
  // 使用者的責任加重，容易出現執行期錯誤。編譯器無法幫上忙。

Stack myStack = new Stack();    // 非泛型運用
MyStack.push(new Integer(4));
MyStack.push(new Double(4.4));
MyStack.push(new String("jjhou"));
System.out.println((String)myStack.pop());  // jjhou
System.out.println((Double)myStack.pop());  // 4.4
System.out.println((Integer)myStack.pop()); // 4

Stack<Integer> iStack = new Stack<Integer>();      // 泛型運用
  // 泛型技術改善了前述缺點，迫使編譯器檢驗元素型別是否爲 Integer。
  // Java 泛型技術正是本文討論重點。
```

Alexander Stepanov/Meng Lee 設計的 STL（Standard Template Library）被納入 C++ 標準之後，泛型技術才終於在實用世界中跨一大步，在資料結構和演算法領域中被廣泛運用。現在，泛型技術也在 JDK1.4 中實現了。

以下我將首先爲你介紹 Java 標準程式庫中負責資料結構和演算法的所謂 Collections Framework，它們原本並不帶有泛型特質。然後我再介紹 JDK1.4 如何使 Collections Framework 帶上泛型特質。

Historical Collection Classes（JDK1.1 之前）

面對容器（資料結構）和演算法領域，Sun 提出一個較 STL 簡略的方案，就是 **Java Collections Framework**。此物在 JDK1.1 之前未達完備，一般又稱爲 **Historical Collection Classes**，提供的容器有 `Arrays`, `Vector`, `Stack`, `Hashtable`, `Properties`, `BitSet`。其中定義出一種走訪群集內各元素的標準方式，稱爲 Enumeration（列舉器）介面，用法如下：

```
// in Java
// 每一個舊式的 collection classes 都提供有一個 enumeration() 或
// elements()，用來給出一個列舉器。
Enumeration enum = ...;
while (enum.hasMoreElements()) {
  Object o = enum.nextElement();
  processObject(o);
}

for (Enumeration enum = ...; enum.hasMoreElements();) {
  Object o = enum.nextElement();
  processObject(o);
}
```

這種手法在 MFC（Microsoft Foundation Classes）中也常看到：

```
// in C++, MFC
void CDocument::UpdateAllViews(...)
    // 巡訪所有的 views
{
    POSITION pos = m_viewList.GetHeadPosition();
    while (pos != NULL)
    {
        CView* pView = (CView*)m_viewList.GetNext(pos);
        ...
    }
}
```

Java Collections Framework（J2SE 之後）

J2SE 之後統稱的 Collections Framework，由三部分組成：

1. 介面（亦即抽象類別），包括 `Collection,List,Set,SortedSet,Map, SortedMap`。
2. 實作類別，包括 `HashSet,HashMap,WeakHeahMap,ArrayList,TreeSet, TreeMap, LinkedList`。它們都可以次第讀寫（*serializable*）和自我複製（*cloneable*）。
3. 演算法（集中於 class `Arrays` 和 `Collections` 之內的一些 static methods），例如：

```
public static void sort(List list);
public static void sort(List list, Comparator comp);
public static int binarySearch(List list, Object key);
public static int binarySearch(List list, Object key,
                          Comparator comp);
public static Object min(Collection col);
public static Object min(Collection col, Comparator comp);
public static Object max(Collection col);
public static Object max(Collection col, Comparator comp);
public static void shuffle(List list);
public static void shuffle(List list, Random rnd);
public static void fill(List list, Object element);
public static void copy(List dest, List src);
```

由於上述容器並未帶有泛型特質，一如先前所言，容器內的異質物件容易帶給程式員困擾，例如：

```
LinkedList myList = new LinkedList();    // non-generic
myList.add(new Double(4.4));             //
myList.add(new String("jjhou"));         //
System.out.println(myList);              // [4.4, jjhou]
System.out.println(Collections.max(myList));
  // ERROR! Exception in thread "main" java.lang.ClassCastException
```

迭代器（Iterator）

Collections Framework 提供一種迥異於以往的群集元素走訪標準介面，稱爲 Iterator（迭代器）介面，遵循 GOF 於《*Design Patterns*》一書所定義的 **Iterator** 設計樣式。用法如下：

```
// in Java
Collection c = ...;
Iterator i = c.iterator();
while (i.hasNext()) {
  process(i.next());
}
```

這和 C++ STL 的用法極爲類似，只不過由於 C++提供運算子多載化能力，所以 STL 直接以 operator++ 實作出迭代器「前進至下一位置（next）」功能，並以 operator* 實作出「提領（dereference）」功能：

```
// in C++
list<int> myList = ...;
list<int>::iterator i = myList.begin();
while (i != myList.end()) {
  process(*(i++));
}
```

條件判斷式（Predicate）

C++ STL 允許我們對動作（演算法）進行某種條件約束。例如我們希望在某個 int 容器的某個區間內計算「數值不小於 40」的元素個數，可以採用 STL 演算法 count_if() 並這麼做：

```
count_if(c.begin(), c.end(),
         not1(bind2nd(less<int>(),40)));  // 不小於 40
```

其中 not1 和 bind2nd 是一種匪夷所思的手法，稱爲配接器（adapters），本文對此並不多做介紹，技術細節請見《STL 源碼剖析》第 8 章。

Java Collections Framework 不提供如此巨大的彈性，但它從另一角度出發，允許我們對迭代器設限，達到某種篩選目的。假設我們希望走訪某個 String 容器，篩選其中「以 "JJ" 開頭」的字串列印出來，我們可以設計一個特殊（帶條件）的迭代器如下：

```
interface Predicate {
  boolean predicate(Object element);      // 條件判斷式，介面。
}

class PredicateIterator implements Iterator { // 條件迭代器
  public PredicateIterator(Iterator iter, Predicate pred) {
    // 接受一個一般迭代器和一個條件判斷式，準備融合成爲一個條件迭代器。
```

```
  }
  public void remove() { ... }       // 迭代器必備功能之一
  public boolean hasNext() { ... }   // 迭代器必備功能之二
  public Object next() { ... }       // 迭代器必備功能之三
}
```

事實很明顯了：我們可以在條件迭代器的 next()和 hasNext()內利用條件式做點手腳。這個條件迭代器的用法如下：

```
public class PredTest {
  static Predicate pred = new Predicate() {
    public boolean predicate(Object o) {
      return o.toString().startsWith("JJ"); // 實作我們自己的條件式
    }
  };
  public static void main (String args[]) {
    List list = Arrays.asList(args);
    Iterator i1 = list.iterator();                    // 一般迭代器
    Iterator i = new PredicateIterator(i1, pred);  // 條件迭代器
    while (i.hasNext()) {
      System.out.println(i.next());
    }
  }
}
```

請注意，Java Collections 的作法是在迭代器上設條件，而 C++ STL 的作法是在演算法上設條件（有時不稱爲條件，而是一種附加運算）。STL 容器所提供的迭代器純粹只是一種指位器（當然你也可以運用「配接（adapt）」技巧來修飾它，唔，這屬於高階技術議題）。由於某些動作無法靠「條件式迭代器」完成，所以某些 Java 演算法也允許帶有條件，例如先前所列的 sort(),binarySearch(), max(),min()都有著帶條件（用以表示大小比較準則）的第二型式。

Java with Generics

Java 爲保持語言的簡單性，強迫程式員自己動手做一些事情：由於 Java 容器的元素型別都是 *Object*，而任何一個 Java 物件都是 *Object* 物件，都可納入容器之內，因此你必須記住你的元素型別；一旦從中取出元素，更進一步處理之前必須先將它轉型，從 *Object* 轉爲其原本型態。

如果以泛型來擴充 Java 語言，就有可能以一種更直接的方法來表現容器的相關資訊，於是編譯器可以追蹤記錄你所擁有的元素型別，而你也就不再需要對取回的

元素做向下轉型動作。這種機制類似 Ada 語言的 **generic**s 或 C++語言的 **template**s。不過 Java 並不提供類似的或新的關鍵字，而是以一種「假型式」呈現。

「泛型爪哇」的研究，世界各處多有進行，其中以 **Generic Java**（**GJ**）獨得 Sun 的青睞，成爲 JSR14[1]的技術基礎。你可以免費下載 GJ[2]，也可以從 Sun 網站免費下載 JDK1.4+JSR14[3]，兩者的一般表現差不多（畢竟同宗），但 GJ 在某些高階主題（例如所謂 **"bound**s"，稍後詳述）上暫勝一籌。本文將同時介紹兩者的安裝與設定，及其具體技術表現。以下當我說 **Java with Generics**，是一種泛稱；如果我說 **GJ** 或 **JDK1.4+JSR14**，便是指特定某個實作品。

Java with Generics 以角括號（< >）標示型別參數，原因是 C++ 程式員對它們比較熟悉，而且其他類型的（大、中、小）括號早都被用上了。Java with Generics 的幾個關鍵特性包括：

- 相容於 Java 語言。Java with Generics 是 Java 的超集。每個 Java 程式在 Java with Generics 中都仍然合法而且有著與過去完全相同的意義。
- 相容於 Java 虛擬機器（JVM）。Java with Generics 被編譯爲 JVM 碼。JVM 不需爲了它而有任何改變。因此傳統 Java 所能執行之處，Java with Generics 都能執行，包括在你的瀏覽器上。
- 相容於既有程式庫。既有的程式庫都能夠和 Java with Generics 共同運作，即使是編譯後的 .class。有時候我們也可以將一個舊程式庫翻新，加上新式型別，而不需更動其源碼。Java collections framework 就是這樣被翻新而加上泛型特質（後述）。
- 高效（efficiency）。泛型相關資訊只在編譯期（而非執行期）才被維護著。這意味編譯後的 Java with Generics 程式碼在目的和效率上幾乎完全和傳統的 Java 程式碼一致。

[1] JSR: Java Specification Requests，是 Java 規格的申請機構。編號第 14 就是泛型議題。

[2] http://www.research.avayalabs.com/user/wadler/gj/

[3] JDK1.4 可自 http://java.sun.com 下載，JSR14 可自 http://jcp.org/jsr/detail/14.jsp 下載。

Generic Java (GJ)

GJ 下載與設定

圖 1 是 GJ 官方網站的入口畫面。從中下載 gjdist1.2.zip，解壓縮後安裝如下，預設置於 C:\GJ：

```
CLASSES          <DIR>          04-12-02
SRC              <DIR>          04-12-02
DOC              <DIR>          04-12-02
GJC      BAT                51   08-05-99
VERSION  TXT             1,152   08-05-99
GJCR     BAT               134   04-12-02
```

其中 CLASSES 子目錄內含 **java.util.Collections** 和 **gjc.Main**，前者是 GJ 提供的（泛型）Collections Framework，後者是 GJ 編譯器主程式。

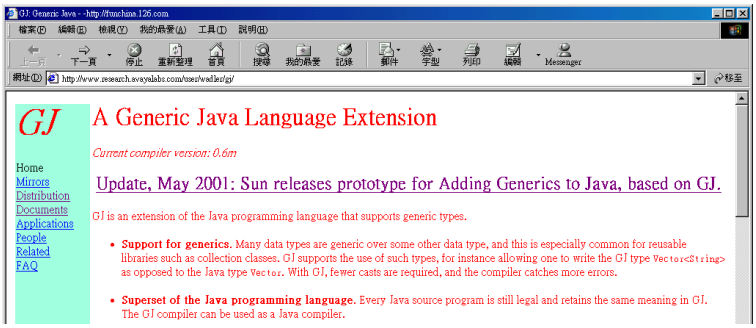

圖 1/ GJ 官方網站的進入畫面

GJ 環境設定

根據 GJ 官方網站上的說明，我們撰寫 **gjcr.bat** 如下，其內是連續一行命令，因篇幅限制而折轉並縮排，以利閱讀：

```
java -ms12m gjc.Main -bootclasspath
  c:\gj\classes\;c:\jdk1.3\jre\lib\rt.jar;
  c:\jdk1.3\jre\lib\i18n.jar %1 %2 %3 %4 %5 %6 %7 %8 %9
```

這個 gjcr.bat 將被用來做爲編譯 GJ 程式時的編譯器外覆批次檔。其意義如下：

- -ms12n，就是一般 Java 編譯器的 -Xms12n，爲的是令編譯器配置更大的 heap，因爲泛型程式的編譯需要較多記憶體。

- `gjc.Main` 是 GJ 編譯器的主程式。
- -bootclasspath 用來改變 core classes（核心類別）的載入次序。這是因爲 JRE（Java 執行環境）在動態載入 classes 時，如果遇到和 core classes 同名者（package+className），會優先載入 core classes。現在既然 GJ 提供了自己的一套 Collections 核心程式庫（就放在 c:\gj\classes 內），爲替換原本 JDK 的那一套，我們必須利用這個選項來設定載入次序。
- 你應該這麼使用本批次檔：`c:\>gjcr Test.java`

GJ 也可以編譯傳統的非泛型 Java 程式，但你必須採用下面這個外覆批次檔：

```
java -ms12m gjc.Main %1 %2 %3 %4 %5 %6 %7 %8 %9
```

下面是我爲 JDK1.3+GJ 做的一份環境設定批次檔：

```
@echo off
rem JDK1.3 with Generic Java (GJ)
rem appending C:\GJ to PATH (as below) is just for gjc(r).bat
set PATH=C:\jdk1.3\bin;C:\WINDOWS;C:\WINDOWS\COMMAND;C:\GJ
set classpath=.;d:\jdk1.3\lib\tools.jar;C:\GJ\classes
```

將這份環境設定檔設爲某個 DOS 視窗的「內容」表單下的「程式」附頁中的批次檔，如圖 2，那麼每當開啓該 DOS 視窗，就會自動設定好上述的 JDK1.3+GJ 開發環境和執行環境，如圖 3。

圖 2/ 將某個 DOS 視窗的批次檔設爲前述之 JDK1.3+GJ 環境批次檔。

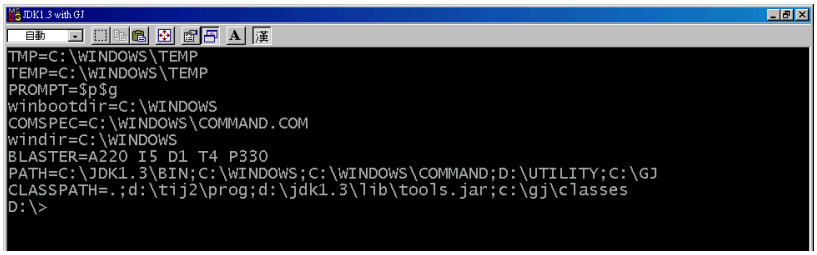

圖 3/ 選按圖 2 所設定的 DOS 視窗，即開啓一個 JDK1.3+GJ 編譯和執行環境

JDK 1.4+JSR14

JDK1.4 下載與設定

圖 4 是 Sun 官方網站的進入畫面。從中下載 JDK1.4 並安裝（我在答問過程中選擇一併下載 Forte，這是個高階 Java 開發工具，與本文主題無關，但會影響安裝路徑），預設置於 C:\J2SDK_Forte\jdk1.4.0。

圖 4/ Sun 官方網站的進入畫面

JSR14 下載與設定

圖 5 是 JSR14 官方網站的進入畫面。可從 http://jcp.org/jsr/detail/14.jsp 下載 JSR14 實作品 adding_generics-1_2-ea.zip，解壓後安裝，預設置於 c:\jsr14_adding_generics-1_2-ea：

```
JAVAC     JAR       448,175   03-13-02  13:05 javac.jar
CHANGES               320   03-13-02  12:54 CHANGES
COPYRI~1            1,201   03-13-02  12:54 COPYRIGHT
LICENSE            10,522   03-13-02  12:59 LICENSE
README              2,374   03-13-02  12:54 README
```

```
COLLECT  JAR        44,392   03-13-02  13:04 collect.jar
JAVAC          <DIR>          04-13-02   3:06 javac
EXAMPLES       <DIR>          04-13-02   3:06 examples
SCRIPTS        <DIR>          04-13-02   3:06 scripts
```

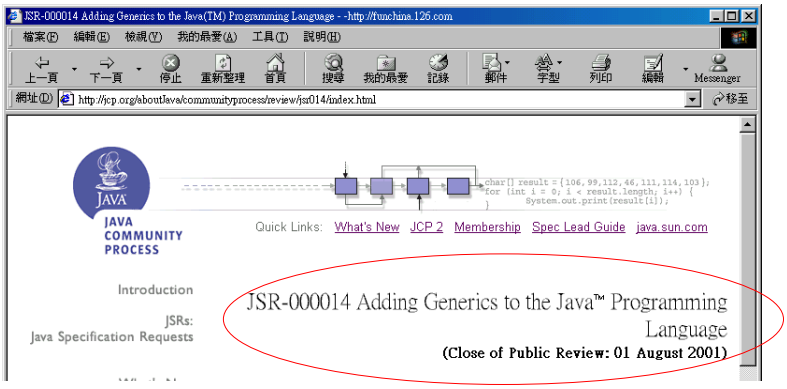

圖 5/ JSR014 官方網站的進入畫面

JDK1.4+JSR14 環境設定

以安裝 JSR14 實作品之後所得的 SCRIPT\javac.bat 爲依據，撰寫一個 **javag.bat** 如下，做爲編譯器的外覆批次檔（爲免和傳統的 javac 同名混淆，我爲它命名爲 javag.bat，其中 g 取義自 generic）：

```
@echo off

:J2SE14
if not exist %J2SE14%\bin\javac.exe goto BADJ2SE14
if not exist %J2SE14%\jre\lib\rt.jar goto BADJ2SE14
goto JSR14DISTR
:BADJ2SE14
echo %J2SE14% does not point to a working J2SE 1.4 installation.
goto end

:JSR14DISTR
if not exist %JSR14DISTR%\javac.jar goto BADJSR14DISTR
if not exist %JSR14DISTR%\collect.jar goto BADJSR14DISTR
goto args
:BADJSR14DISTR
echo %JSR14DISTR% does not point to a working JSR14 installation.
goto end

:args
```

```
if not "%1" == "" goto compile
%J2SE14%\bin\javac -J-Xbootclasspath/p:%JSR14DISTR%\javac.jar
goto end

:compile
%J2SE14%\bin\javac -J-Xbootclasspath/p:%JSR14DISTR%\javac.jar -
bootclasspath %JSR14DISTR%\collect.jar;%J2SE14%\jre\lib\rt.jar -gj
-warnunchecked %1 %2 %3 %4 %5 %6 %7 %8 %9

:end
```

這個批次檔要求兩個環境變數 **J2SE14** 和 **JSR14DISTR**（應分別指向 JDK1.4 和 JSR14 的安裝目錄），然後首先檢查 JDK1.4 的兩個重點檔案是否存在，再檢查 JSR14 的兩個重點檔案是否存在，然後再檢查是否有命令列參數，最後在標籤 `:compile` 處執行 javac 編譯器，令 JRE 在載入 core classes 時優先考慮 JSR14 所供應的 **javac.jar** 和 **collect.jar**，並加上 **`-gj`** 選項啓動泛型特性，加上 **`-warnunchecked`** 選項以求面對非泛型容器時給予警告。

最好是把這個 javag.bat 放在 JDK1.4 的 bin 目錄下，以便在任何 JDK1.4+JSR14 環境下都能被喚起。爲了更方便設定環境，我撰寫一份環境設定批次檔如下：

```
@echo off
rem JDK1.4 + JSR14
set JSR14DISTR=c:\jsr14_adding_generics-1_2-ea
set J2SE14=c:\J2SDK_Forte\jdk1.4.0
set PATH=%J2SE14%\bin;C:\WINDOWS;C:\WINDOWS\COMMAND
set classpath=.;%J2SE14%\lib\tools.jar
```

現在，如圖 2 所示，將這個環境設定檔設爲某個 DOS 視窗的批次檔，於是每當開啓該 DOS 視窗，就會自動設定好 JDK1.4+JSR14 環境，如圖 6。

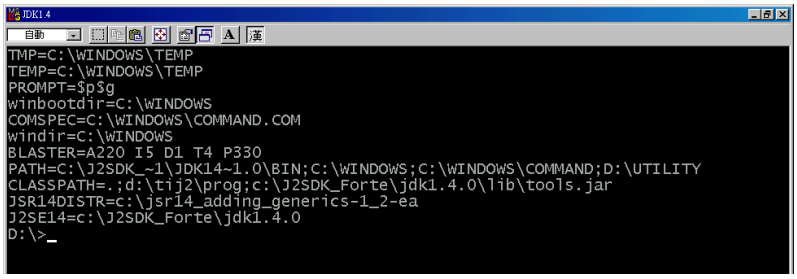

圖 6/ 選按設定後之 DOS 視窗，即開啓一個 JDK1.4+JSR14 編譯環境。

Java with Generics 實例探討

泛型容器的運用

現在我們有了兩個可以實現 Java with Generics 的環境，一個是 JDK1.3+GJ，另一個是 JDK1.4+JSR14。下面是關於容器的一些測試，程式碼自帶註解。

```
LinkedList<Integer> il = new LinkedList<Integer>();
il.add(new Integer(0));
il.add(new Integer(1));
il.add(new Integer(5));
il.add(new Integer(2));

Integer maxi = Collections.max(il);       // Algorithm
System.out.println(maxi);                 // 5
Collections.sort(il);    // Algorithm
System.out.println(il);  // [0, 1, 2, 5]
```

以上指定 il 是個內含 Integer 元素的 LinkedList。加入 4 個數值，然後運用演算法 max()找出最大值，並運用 println()直接列印整個容器內容。

我以類似動作施行於LinkedList<String>, LinkedList<LinkedList<String>>, ArrayList<Double>, Vector<Character>, HashSet<String>,TreeSet<Long>, HashMap<Integer,String>, TreeMap<Integer,String> 身上，並根據其特性分別運用演算法 max(),min(),sort()，都能順利運作。例如：

```
// 以下以 Integer 爲鍵值，String 爲實值。
TreeMap<Integer, String> istm = new TreeMap<Integer, String>();
istm.put(new Integer(3), new String("jjhou"));
istm.put(new Integer(1), new String("jason"));
istm.put(new Integer(9), new String("jamie"));
istm.put(new Integer(7), new String("jiang"));
System.out.println(istm);
  // {1=jason, 3=jjhou, 7=jiang, 9=jamie}
```

上述的 TreeMap 是一種有自動排序能力的容器（隸屬 SortedMap 介面），所以元素安插進去後不需呼叫 sort()即井然有序（以元素的鍵值做升冪排序）。

所有測試動作均納入本文所附程式 Test.java 中，此處不一一列出。

使用者自定型別（user-defined types）

現在我們來試試使用者自定型別。假設我需要一個 class Empolyee 用來表現我的職員人事資料，並以 TreeSet 來容納所有這些資料。由於 TreeSet 會自動排序，因此它必須知道 Employee 物件如何比較大小。爲此我讓 class Employee 實作出 Comparable 介面（並因而必須設計 compareTo()）。傳統（非泛型）的寫法是：

```
public class Employee implements Comparable {
  public int compareTo(Object obj) {
    Employee emp = (Employee)obj;     // 必須先轉型
    ... // 這裡決定如何比較大小
  }

Employee empv[] = {
  new Employee("Finance", "Degree, Debbie"),
  new Employee("Engineering", "Measure, Mary"),
};
Set emps = new TreeSet(Arrays.asList(empv));
Employee maxEmp = (Employee)Collections.max(emps);
  // 演算法 max()也需要知道 Employee 如何比大小，
  // 它會呼叫 Employee.compareTo()
```

由於缺少型別自動管控，程式員比較容易出錯。一旦 Java 支援泛型特性，我們可以改而這麼寫：

```
public class Employee implements Comparable<Employee> {
  public int compareTo(Employee emp) {
    ... // 這裡決定如何比較大小。不必先有轉型動作。
  }

Employee empv[] = {
  new Employee("Finance", "Degree, Debbie"),
  new Employee("Engineering", "Measure, Mary"),
};
Set<Employee> emps = new TreeSet<Employee>(Arrays.asList(empv));
Employee maxEmp = Collections.max(emps);
```

泛型容器的設計

下面運用泛型手法設計一個 Queue。由於內部使用 LinkedList，所以整個實作非常簡短。原則很單純，只要在元素型別出現處，一律將元素型別改爲未定型別 T 即可。任何一個符號（如本例的 T）只要在 class 宣告式中被含括於角括號內，就被編譯器視爲一個未定型別：

```
import java.util.*;
public class JQueue<T>
{
  protected LinkedList<T> mySequence;

  public JQueue() {
    mySequence = new LinkedList<T>();
  }
  public boolean isEmpty() {
    return mySequence.isEmpty();
  }
  public int size() {
    return mySequence.size();
  }
  public boolean add(T obj) {
    return mySequence.add(obj);
  }
  public void push(T obj) {
    mySequence.addFirst(obj);
  }
  public synchronized T pop() {
    return mySequence.removeLast();
  }
  public synchronized String toString() {
    return "Queue( " + mySequence.toString() + " )";
  }
}
```

JQueue 的運用如下：

```
JQueue<Employee> eq = new JQueue<Employee>();
eq.push(new Employee("Finance", "MJChen"));
eq.push(new Employee("Engineering", "JJHou"));
eq.push(new Employee("Sales", "Grace"));
eq.push(new Employee("Support", "Jason"));
System.out.println(eq);
   // Queue( [[dept=Support,name=Jason], [dept=Sales,name=Grace],...
System.out.println(eq.pop());     // [dept=Finance,name=MJChen]
System.out.println(eq.pop());     // [dept=Engineering,name=JJHou]
System.out.println(eq.pop());     // [dept=Sales,name=Grace]
System.out.println(eq.pop());     // [dept=Support,name=Jason]
```

泛型函式（泛型方法，Generic Method）

Java 所稱的 method，就是 C/C++ 所稱的函式。若譯爲「方法」易與一般中文混淆，喪失術語的獨特性；不譯又恐到處中英夾雜，視覺效果不佳。故本文將 Java method 一貫稱爲函式。

如果函式擁有自己的型別參數，我們稱它爲一個泛型函式。下面是 Java 泛型函式的實例，請注意型別參數 `<T>` 的出現位置：

```
// assume in class Test
public static <T> T gm (List<T> list)
{
  T temp = list.iterator().next();   // get the first one
  return temp;                       // and just return.
}
```

下面是 Java 泛型函式的運用實例，此時會以 `Employee` 取代上述的 `T`：

```
LinkedList<Employee> empList = new LinkedList<Employee>();
... // add some elements
System.out.println(Test.gm(empList));
```

受限型別參數（Bounded type parameter）

型別參數如果必須實作出某個已知介面，或必須是某已知 class 的 subclass，我們稱此爲一個 "**bounded**"（受限的）型別參數，而該限制條件（某個或某些 Java interfaces 或 classs）則稱爲 "**bounds**"。在 C++中，如果型別 A 被傳入泛型函式之後，無法滿足泛型函式內對 A 物件的所有運算（例如泛型函式中對 A 物件進行了 `+,-,*,/`，而型別 A 無法完全滿足所有這些運算），那麼編譯器會報錯。Java 的作法顯然比較更先進些，允許我們在宣告之時就將限制條件明白列出，這對程式維護比較有利。下面是個例子：

```
// assume in class Test
public static <T implements Comparable<T>> T gm (List<T> list)
{
  T temp = list.iterator().next();   // get the first one
  return temp;                       // and just return.
}
```

以上函式宣稱，`gm()` 接受一個 `List`，其內的元素型別都是 `T`；回傳一個元素，

型別亦爲 T。最前面的角括號內宣告了型別參數 T，並指出 T 必須實作出 Comparable<T>（不能只是 Comparable）介面。面對呼叫動作如下（一如先前的泛型函式測驗）：

```
LinkedList<Employee> empList = new LinkedList<Employee>();
... // add some elements
System.out.println(Test.gm(empList));
```

編譯器推導出 gm()標記型式中的型別參數 T 必須被具現化爲 Employee，編譯器亦檢驗出 class Employee 確實實作了 Comparable<Employee>，因而得以順利讓它通過。下面是 "**bounded**"（受限）型別參數的另一個例子：

```
public class Hashtable<Key extends Object, Data extends Object>
{
    private static class Entry<Key, Data> {
    Key key;
    Data value;
    Entry<Key, Data> next;
    ...
}
```

一般而言導入"**bounds**" 的方式是，在型別參數之後寫上 "implements" 再加一個 interface 名稱，或是在型別參數之後寫上 "extends" 再加一個 class 名稱。不論在 class 標頭或泛型函式標記式中，凡型別參數可以出現的地方，**bounds** 都可以出現。bounding interface 或 bounding class 本身還可以被參數化，甚至形成遞迴，例如前例的 **bound** Comparable<T> 就內含了 bounded（受限）型別參數 T。

我個人的測試經驗顯示，JDK1.4+JSR14 尚無法接受「**bounded** 型別參數」，GJ 才可以。稍後有一些關於 **bounds** 的驗證。

次第讀寫（Serialization）

Java Collections Framework 較諸 C++ STL 的一個極大優勢就是：它支援物件永續（**Object Persistence**）。這是一個大而重要的主題，輕量級的作法是所謂的次第讀寫（**Serialization**），也就是「以某種次序寫入，以相同次序讀出」。Java Collections 之所以能夠獲得這種優勢，在於整個 Java 標準程式庫是個龐大的單根（single-root）繼承體系，從這個角度切入，就有了很好的憑藉點可以製作出物件永續性。C++ MFC

也是因爲憑藉了一個單根繼承體系而能夠製作出物件永續性（技術細節請參考《多型與虛擬 2/e》第 6 章（by 侯捷，開放於侯捷網站）。

針對物件永續，我測試了前述提及的各種泛型容器，包括 `LinkedList<String>`, `LinkedList<LinkedList<String>>`,`ArrayList<Double>`, `Vector<Character>`, `HashSet<String>`,`TreeSet<Long>`,`HashMap<Integer,String>`,`TreeMap<Integer,String>`。每個（泛型）容器都是一個物件，因此就像對待一般 Java 物件一樣，你可以將整個容器輕易寫入檔案，再輕易讀回來：

```
ObjectOutputStream out =
  new ObjectOutputStream(
    new FileOutputStream("collect.out"));
// 以上是一種典型的 Decorator 設計樣式。

out.writeObject(il);  // 將整個容器（LinkedList<Ingeter>）寫至檔案
out.writeObject(sl);  // 將整個容器（LinkedList<String>）寫至檔案
...
out.close();  // 藉由關閉的動作掃清（flush）output stream.

ObjectInputStream in =
  new ObjectInputStream(
    new FileInputStream("collect.out"));
// 以上是一種典型的 Decorator 設計樣式。

LinkedList il2  = (LinkedList)in.readObject();
LinkedList sl2  = (LinkedList)in.readObject();
...
// 以下讀取動作，次序必須完全相同於塗寫動作。
```

這其中，容器的型別參數必須支援 `Serializable` 介面，才能滿足次第讀寫過程中的需求。例如前述的 class `Employee` 應該改爲這樣：

```
public class Employee implements Comparable<Employee>, Serializable
{ ... }
```

Java with Generic 之實作技術探討

C++採用膨脹法（expansion）

面對 templates，C++ 編譯器的作法是：程式出現多少種型別參數，就產生多少份 template「版本」。例如編譯器看到以下三份運用，就爲 class template `list` 產生三個版本（三份實體），分別是 `int` 版、`double` 版和 `string` 版（相當於我們自

已手寫三個 classes 一樣）：

```
list<int> iList;       // 產生 list 的 int 版本
list<double> dList;    // 產生 list 的 double 版本
list<string> sList;    // 產生 list 的 string 版本
```

這種所謂的**膨脹法**（**expansion**），導致程式碼體積膨脹。但膨脹不是最大的問題，最大問題在於 template 可能被定義於 A 檔案中而被 B 檔案使用，因此膨脹法所引起的錯誤很難被偵測出來，直到聯結期才有所記錄，而且往往難以回溯。

Java 採用擦拭法（erasure）

Java 就不同了，其泛型編譯器的工作是把泛型程式碼譯回一般的非泛型程式碼。這個翻譯程序僅只是「消除型別參數」並「適當加上轉型動作」。例如它把 *List<Byte>* 譯回 *List*，並在必要地點將 *Object* 轉型爲 *Byte*，如此而已。獲得的結果就像在非泛型情況下所寫的 Java 程式。這就是爲什麼我們能夠輕易爲 Java with Generics 和既有的（傳統的）Java 程式庫建立介面的原因，也是爲什麼 Java with Generics 能夠和傳統 Java 擁有相同效率的原因。Java 泛型編譯器保證任何一個被它加入的轉型動作都不會導致錯誤。在這種保證之下，由於泛型編譯器將程式碼翻譯爲 JVM byte codes，所以 Java 平台原本擁有的安全性（safety）和防護性（security）也都獲得了保留。

爲了將泛型碼翻譯爲一般的非泛型碼，編譯器必須爲每個型別做一種特殊的**擦拭**（**erasure**）動作。正是這樣的擦拭動作，才能讓一個「根據泛型完成的 Java 程式」和一個「非泛型的傳統 Java 程式庫」放在一起編譯（因爲兩者的本質一致）：

- 一個參數化型別擦拭後應該去除參數（於是 *List<T>* 被擦拭成爲 *List*）
- 一個未被參數化的型別擦拭後應該獲得型別本身（於是 *Byte* 被擦拭成爲 *Byte*）
- 一個未受限的（unbounded）型別參數擦拭後的結果爲 *Object*（於是 *T* 被擦拭後變成 *Object*）
- 一個受限的（bounded）型別變數擦拭後的結果爲其 **bound** 的擦拭結果（於是 *T implements Comparaible<T>* 便被擦拭爲 *Comparable*）
- 如果某個函式呼叫的回傳型別是個型別參數，編譯器會爲它安插適當的轉型動作（於是 *T implements Comparaible<T>* 便被擦拭爲 *Comparable*）

拭去法驗證

爲了驗證上述的擦拭法則，我們必須檢驗 .class 檔。任何一本討論 JVM 的書籍都會提到 .class 檔案格式。我從 http://www.mcmanis.com/~cmcmanis/java/dump/ 下載了一個 "dumpclass"，這個工具可以協助我分析 .class 檔案內容。以下我以圖 7,8,9 分別比較先前例子的 java 檔和 class 檔。

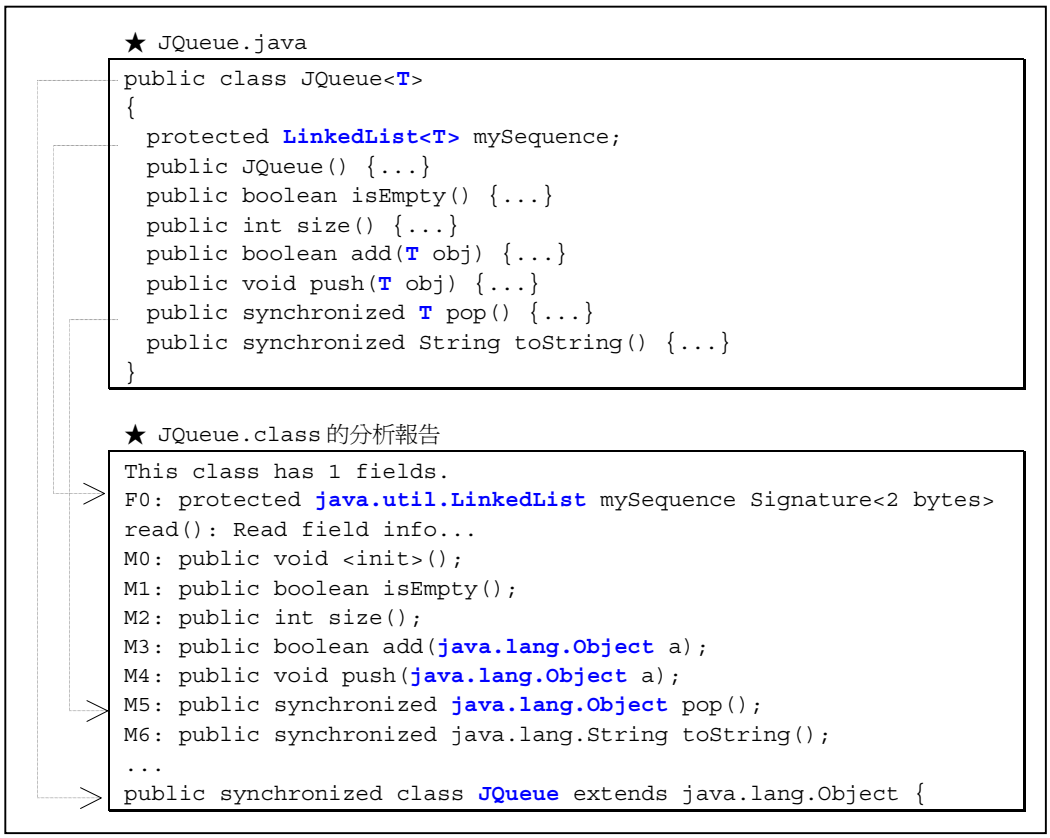

圖 7/ JQueue.java 和 JQueue.class 的比較

圖 7 顯現的擦拭原則是：

- 參數化型別經過擦拭後應該去除參數（於是 *LinkedList<T>* 變成 *LinkedList*）
- 未受限的型別參數擦拭後變成 *Object*（於是 *T* 變成 *Object*）

★ Test3.class（**JDK1.3+GJ** 編譯結果）的分析報告

```
This class has 1 fields.
F0: static java.util.LinkedList empl Signature<2 bytes>
read(): Read field info...
M0: public void <init>();
M1: public static void main(java.lang.String a[]);
M2: public static java.lang.Comparable gm(java.util.List a);
M3: static void <clinit>();
...
```

★ Test3.java

```
import java.util.*;     // for Iterator
public class Test3 {
  static LinkedList<Employee> empl = new LinkedList<Employee>();

  public static void main(String[] args) {
     empl.add(new Employee("Finance", "Degree, Debbie"));
     Empolyee temp = Test3.gm(empl);
  }

  public static <T implements Comparable<T>> T gm (List<T> list)
  {
     return list.iterator().next();
  }
}
```

★ Test3.class（**JDK1.4+JSR14** 編譯結果）的分析報告
（注意，Test3.java 中的 implements Comparable<T> 必須拿掉才能通過編譯）

```
This class has 1 fields.
F0: static java.util.LinkedList empl Signature<2 bytes>
read(): Read field info...
M0: public void <init>();
M1: public static void main(java.lang.String a[]);
M2: public static java.lang.Object gm(java.util.List a);
M3: static void <clinit>();
...
```

圖 8/ Test3.java 和 Test3.class 的比較

圖 8 顯現的擦拭原則是：

- 參數化型別擦拭後應去除參數(於是 *LinkedList<Employee>* 變成 *LinkedList*, *List<T>* 變成 *List*)

- 受限的型別參數擦拭後變成其 **bound** 的擦拭結果（於是 `gm()`的 *T* 被擦拭爲 *Comparable<T>*，而後者又被擦拭爲 *Comparable*）
- 如果某個函式呼叫的回傳型別是個型別參數，編譯器會爲它安插適當的轉型動作（於是 `Empolyee temp = Test3.gm(empl)` 會被編譯器改爲 `Empolyee temp = (Employee)Test3.gm(empl)`，不過這個動作在 dumpclass 的分析報告中顯現不出來）

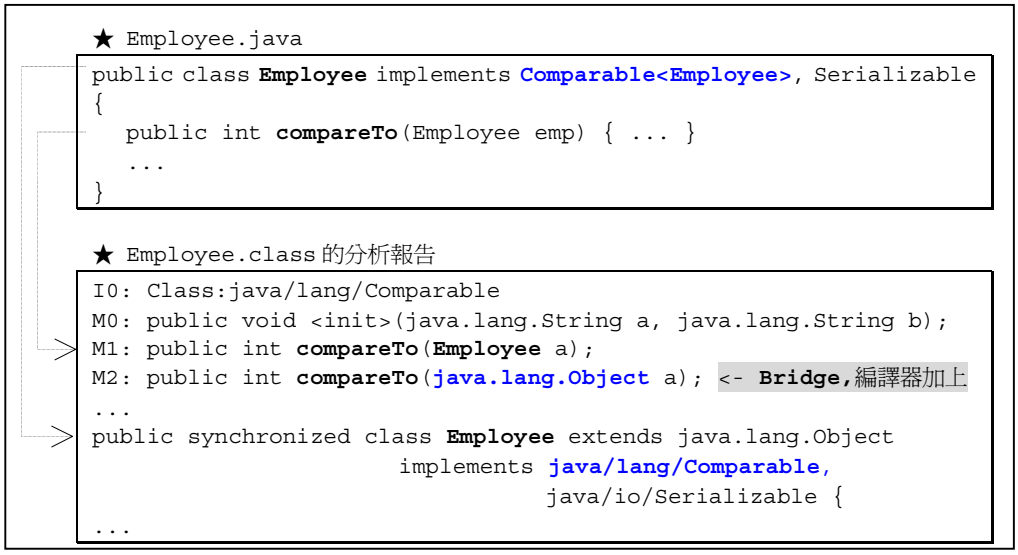

圖 9/ Employee.java 和 Employee.class 的比較

圖 9 顯現的擦拭原則是：

- 參數化型別擦拭後應去除參數（於是 *Comparable<Employee>* 變成 *Comparable*）
- 編譯器對 **compareTo()** 做出兩個版本，第一版本按程式碼的指示接受 *Employee*，第二版本由編譯器自動產出，接受 *Object*，其內動作幾乎可以確定是：`return this.compareTo((Employee)a)`，也就是呼叫第一版本。如此便可保證第一版本一定以正確型式被呼叫，否則會發生執行期錯誤。這是泛型環境下編譯器對使用者的一個貼心服務。多出來的第二版本扮演橋樑的角色，稱爲 **Bridge**。

翻新（Retrofit）

雖然 Java 泛型編譯器的作用是對泛型程式進行擦拭動作，使它還原爲傳統的（非泛型）Java 程式，但如果只是這樣，泛型 Java 對使用者（程式員）又有何意義可言？程式員要的不就是編譯期的檢查和警告工作嗎？如果你使用舊式的 Collections Framework，.class 檔案內沒有任何新式（型別）資訊，編譯器又如何爲你的泛型程式執行型別正確性檢驗？

爲了讓編譯器擁有得以憑藉的資訊，舊式 Collections Framework 不能再用。但若爲了支援泛型而從根本上改變 .class 的結構，又可能影響（新舊 JDK 版本之間）的程式移植性。Java 是動態連結系統，這個問題將十分嚴重。幸運的是 .class 檔案格式允許某種擴充：添加額外的「型別標記（type-signature）」，而這些添加物又可在執行期被 JVM 忽略，保留回溯相容性。

由於只需對 .class 動手腳，而且型式十分固定，所以即使你手上只有 .class files 而無原始碼，也可以將它「泛型化」。假設你有個 LinkedList.class，內含舊式 Java `LinkedList` class，但你希望以泛型方式來使用它。基於日後編譯器所需的型別標記，你必須爲它進行必要的翻新：

```
//新寫一個翻新檔（retrofitting file）如下（各個 methods 只有宣告而無實作）：
class LinkedList<T> implements Collection<T> {
  public LinkedList ();
  public void add (T elt);
  public Iterator<T> iterator ();
}
```

運用 **`-retrofit`** 選項編譯它，編譯器便會取出原本的（非泛型的）LinkedList.class，檢查其型別標記是否等同於「上述翻新檔的型別標記被擦拭後」的結果，如果是，就產生一個帶有新式（泛型）標記的新的 LinkedList.class。舊檔和新檔可以同名，只要 package 不同就行。

根據文獻顯示，Java2 的整個 Collections Framework 已經以此方式翻新（不過我卻在 GJ 和 JSR14 中發現重新寫過（而非翻新）的 Collections classes），程式庫中的每一個 public interfaces, classes, methods 都有適當對應的泛化型別。翻新後的 .class 與原先版本之差異只在於新增的泛型標記，而它們會於執行期間被 JVM 忽略，所

以你可以將此結果執行於 Java2 相容瀏覽器（JVM）中，毫無問題。

圖 10 顯示我對新舊 .class 檔的測試。我寫了一個 JQueue<T> 和一個 NQeueu，兩者行爲完全相同，只不過前者爲泛型，後者爲非泛型（並以非泛型編譯器編譯之，代表舊式 Colections）。然後在應用程式中分別以泛型和非泛型方式來運用它們，獲得結果列於圖 10。圖中 CH[o] 表示編譯期應有的檢驗都有（但不表示一定通過編譯），R[o] 表示執行沒有問題，R[x] 表示無法執行（意指根本沒通過編譯）。我們看到，當應用程式以泛型方式來運用舊式的 NQueue，程式根本編譯不過；如果將 NQueue 以泛型方式重寫（此處以 JQueue<T> 代表），使用上就沒有問題。

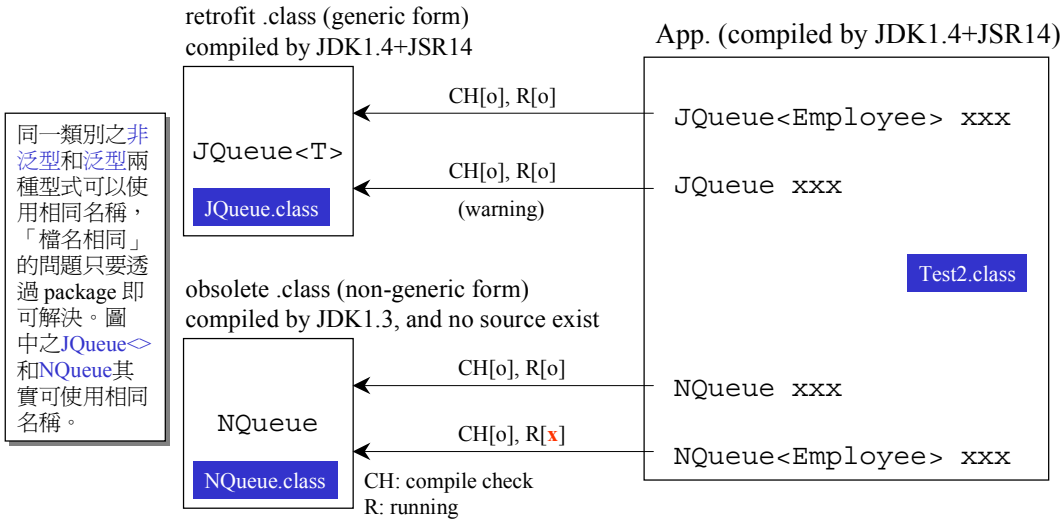

圖 10/ 應用程式分別以泛型和非泛型方式來運用新式和舊式 Collections

我的翻新經驗

爲了更進一步驗證翻新過程，我打算將圖 10 的 NQueue.class（代表舊式 Collections）加以翻新，於是寫一個翻新檔如下：

```
// File: NQueue.java (retrofitting file)
// compiled by JDK1.4: javag -retrofit xxx -d xxx

import java.util.*;
public class NQueue<T>
{
```

```
  protected LinkedList<T> mySequence;
  public NQueue();
  public boolean isEmpty();
  public int size();
  public boolean add(T obj);
  public void push(T obj);
  public synchronized T pop();
  public synchronized String toString();
}
```

GJ 編譯器和 JDK1.4+JSR14 編譯器都有這樣的選項：

```
-retrofit <pathname>  Retrofit existing classfiles with generic types
-d <directory>        Specify where to place generated class files
```

於是我在 JDK1.4+JSR14 編譯環境中執行以下命令：

```
D:\javacol\prog\generics\jdk14>javag NQueue.java -verbose
-retrofit com\jjhou\util -d .
[parsing started NQueue.java]
[parsing completed 220ms]
[loading c:\J2SDK_Forte\jdk1.4.0\jre\lib\rt.jar(java/lang/Object.class)]
[loading c:\jsr14_adding_generics-1_2-ea\collect.jar(java/util/LinkedList.class)]
[loading c:\jsr14_adding_generics-1_2-ea\collect.jar(java/lang/String.class)]
[checking NQueue]
[total 6210ms]
[retrofitting NQueue]
error: cannot access NQueue
file NQueue.class not found
```

我預想 Java 泛型編譯器會編譯目前目錄下的 NQueue.java（翻新檔），並將擦拭結果拿來和（某個 classpath 之下的）com.jjhou.util.NQueue.class 比對，如果比對正確就輸出一個新的 NQueue.class 置於目前目錄。而我的舊式 NQueue.class 的確放在 d:\tij2\prog\com\jjhou\util 之中，並且設了一個 classpath 爲 d:\tij2\prog。但錯誤訊息顯示，編譯器找不到「待被翻新的」（舊式）.class 檔案。可能是因爲我還不夠了解如何對 `-retrofit <pathname>` 做正確的設定。目前我尚未能夠解決這個疑惑。

總結

- C++ STL 以泛型技術（Generics）完成資料結構和演算法，獲得很大的成功。
- Java 傳統以來即有所謂的 Collections Framework 提供處理資料結構和演算法。
- 新式編譯器（GJ 或 JDK1.4+JSR14）可以處理 Java 泛型語法。

- Java 泛型語法並未提供新關鍵字，而是以角括號（<>）表示型別參數。
- Java 編譯器以擦拭法處理泛型語法，將泛型語法還原爲舊式的非泛型語法。此與 C++ 面對泛型所採用的膨脹法不同。
- 爲了編譯期的型別檢驗需求，舊式 Collections Framework 必須加入新式泛型型別標記（Type Signature）。方法之一是將舊式 classes 重新寫過，方法之二是採用翻新（retrofitting）法。
- 泛型 Java 的型別參數，可以帶有「條件」— 是即所謂 **bound**s。這是 C++ STL 做不到的。
- Java Collections Framework 帶有物件永續（次第讀寫，Serialization）功能，自古即有，泛型時代亦然。關鍵在於其標準程式庫是個單根系統 — MFC 亦然，所以 MFC Collections 也有相同能力。這卻是 C++ STL 未能做到的。
- 泛型 Java 並沒有帶來任何 Java 本質改變，只是給予程式員在型別檢驗上的協助，使程式員的工作更輕鬆，更不易出錯。

更多資訊

- *GJ : A Generic Java, java may be in for some changes*, Philip Wadler, DDJ, Feb., 2000。截至目前我認爲最重要的一篇「通俗的」「泛型爪哇」技術文章。
- *Thinking in Java* 2/e, Bruce Eckel, Prentice Hall, 2000。第 9 章 *Holding your data* 對於 Java Collections 有廣泛的介紹，第 11 章 Java I/O 和第 12 章 RTTI 都有極爲難得而深入的技術表現。
- *Java Collections, Comprehensive coverage of the Java Collections Framework*, John Zukowski, Apress, 2001。全書介紹 Java Collections，在資料的收集、整理、說明上有良好表現。附錄 C 也談到了「泛型爪哇」的發展概況。