



Object Serialization 機制

侯捷

資訊工作者 / 專欄執筆 / 大學教師
jjhou@jjhou.com <http://www.jjhou.com>

java



講題簡介



只要 objects 儲存於磁碟或傳輸於網絡，就需要用到 **Serialization** 技術。Java serialization 包裝得非常好，程式員只需 "**implements Serializable**"，幾乎不必再付出其他努力，便可充份享用 **object persistence** 能力。

本講題告訴您 **Serializable** 背後的故事，帶您認識 **Serialization** 的技術關鍵以及 **Java** 的實現手法。並具體讓您知道 **object** 被儲存起來的模樣。同時也告訴您如果不使用預設機制而自行實現 **serialization**，會帶來什麼隱微影響。



Object Serialization 相關源碼

欲透徹理解 **Java** 在物件技術上的優異表現並發揮於你的工作，應當對以下三個領域有堅實的基礎。本研討主題（**Object Serialization**）正是橫跨這三個領域：

- **java.io**
- **java.util**
- **java.lang (.reflect)**

➤ Java programming 方面的重要著作：《*Effective Java*》by Joshua Bloch，全書主要便是談論 java.io, java.util, java.lang。



什麼是Object Serialization

(物件序列化、物件次第讀寫)

- 針對 **object** 或 **a graph of objects** (**a web of objects**) 產生一個 **serialized representation** (一連串的連續性表述) 。
- 物件的 **values** 和 **types** 都以足夠的資訊 (信息) 被 **serialized**，確保等價的 (**equivalent**) **object** 可被重建 (**recreated**) 。
- 重建動作又稱為 **deSerialization** 。



什麼是Object Serialization

: 進一步說明

Java Serialization 允許你將任何實現 **Serializable interface** 的物件轉變成一連串帶有次序性的 **bytes** (a sequence of bytes) 。並於日後完整地回復(**fully restored**) 而重新生成原物件(**regenerate the original object**) 。

可應用於物件永續或網絡傳輸；允許每個物件定義其外部格式(external format)



以編程方式控制Serialization

- Serialization 相關設施（facilities）包括interfaces, methods, keywords

程式中可控制 **Serialization** 的工具：

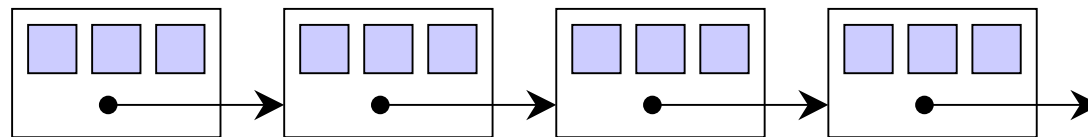
- **java.io.Serializable**: 對於這種物件, stream 包含足量資訊用以將 stream 中的 fields 恢復為 class 的一個相容版本
- **java.io.Externalizable**: 對於這種物件, class 僅需對其內容的外部格式（external format）承擔責任。
- **writeExternal()**, 必要性地搭配 **java.io.Externalizable** 使用。
- **readExternal()**, 必要性地搭配 **java.io.Externalizable** 使用。
- **transient**, 表示某個 field 不會被 default serialized.
- **writeObject()**, 選擇性地搭配 **java.io.Serializable** 使用。
- **readObject()**, 選擇性地搭配 **java.io.Serializable** 使用。

- 關於 Serialization，如果是 Class, ObjectOutputStream, String 和 array 的物件，需要特殊處理。其他任何物件都必須實現 Serializable 或 Externalizable，才能被寫至/讀自 stream。
- 經查驗，java.util 之中沒有任何classes使用 Externalizable。



Serialization是一種深層拷貝 (deep copy)

Serialization不只儲存物件的**image**（映件、映像、化身），還儲存物件中指涉（參考、引用、**refer**）的所有物件，以及那些物件所指涉（參考、引用、**refer**）的所有物件...



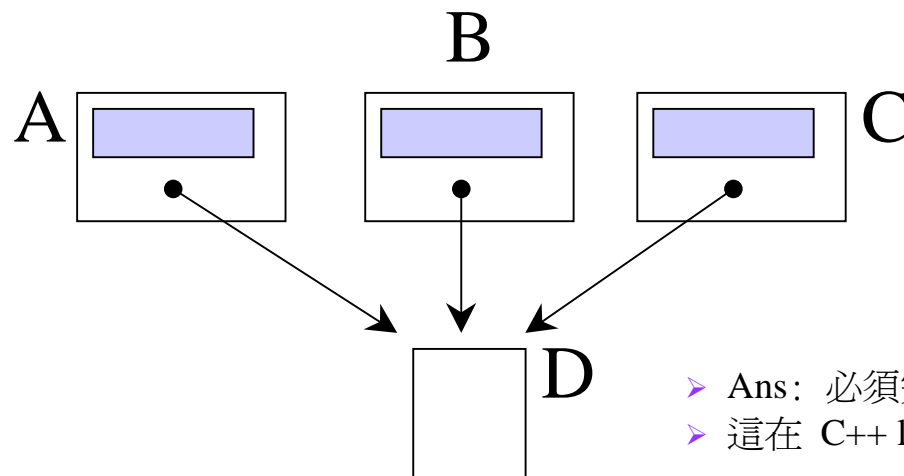
➤ 任何物件的第一個 referenced 會導致該物件被 serialized 並被給予一個 handle。後續再有對該物件的 reference 便會被編碼為該 handle。



Serialization 之於 reference semantics

➤ 相對於 value semantics

如果你 **serialize A,B** 兩物件，而它們都有個 **reference** 指向第三個（同一個）**C**物件，會發生什麼事？當你回復 (**restore, deSerialize**) **A,B** 兩物件時，是否只獲得一份(**one occurrence**) **C** 物件？



- Ans: 必須完全恢復原狀
- 這在 C++ library (例如MFC) 需要大費周張



完整系統狀態的保存 (Serialized the state of a system)

如果你要儲存系統狀態(the state of a system)，最安全的作法是以一個“**atomic**” operation來完成 **serialization** 動作。如果你 **serialize** 某些東西，然後做某些動作，然後再 **serialize** 另一些東西，你將無法獲得保證能夠安全回復你的系統。最好是將構成你的系統狀態(the state of your system)的所有物件放進**單一容器**內，並以**單一動作**將該容器寫出；然後你便能夠以**單一函式**回復它。

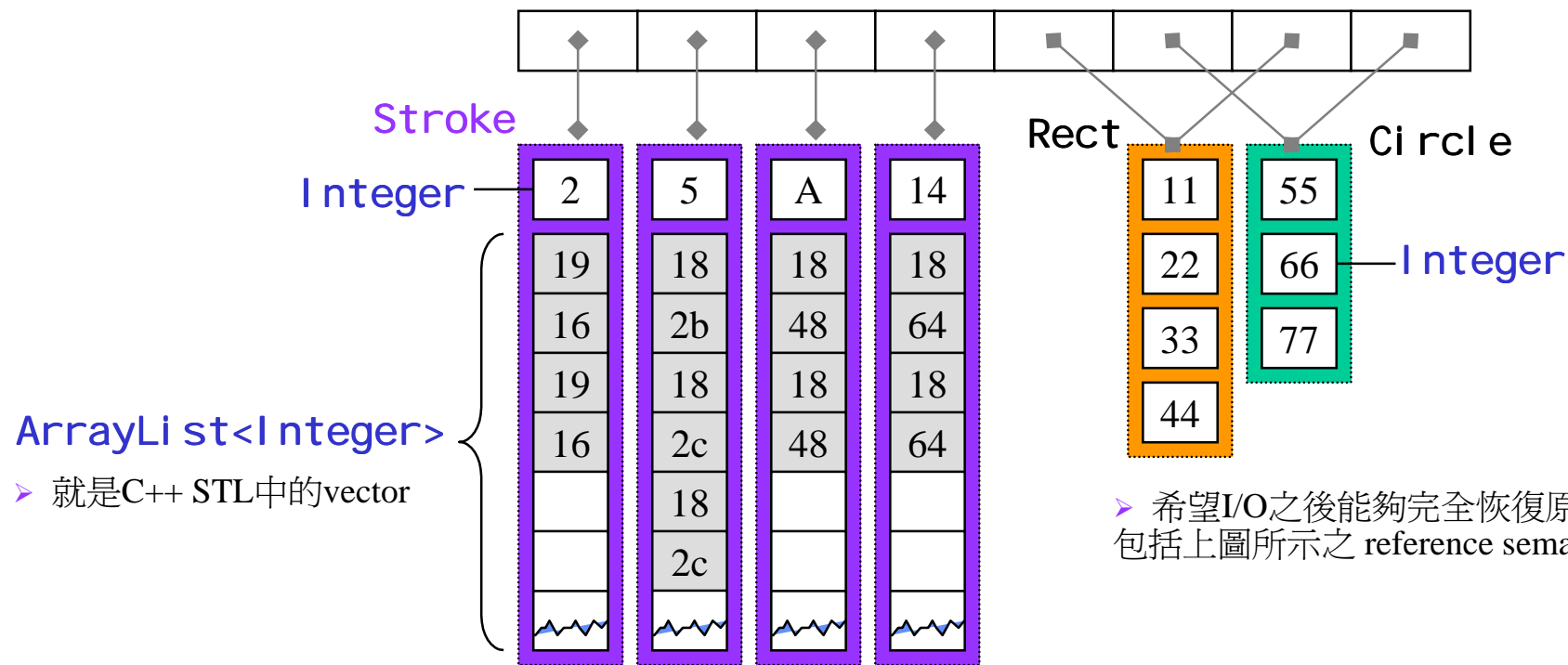


Serialization 實例

: 一群有組織的物件

- 都是 object references，以pointer-like 方式畫出
- 本例用到的classes 計有LinkedList, Stroke, Integer, ArrayList, Rect, Circle，稍後皆有說明。

LinkedList<Object>





Serialization 實例

: 備妥 collection 並添加 Stroke 元素

➤ class Stroke 稍後說明

```
// 也可以把所有的 <> 都拿掉。<>表現出Java Generi c技術。
```

```
Li nkedLi st<Obj ect> sl = new Li nkedLi st<Obj ect>();
```

```
ArrayLi st<I nteger> ia = new ArrayLi st<I nteger>();
```

```
ia.add(new Integer(0x19));      ia.add(new Integer(0x16));
```

```
ia.add(new Integer(0x19));      ia.add(new Integer(0x16));
```

```
sl.add(new Stroke(new Integer(2), ia));
```

```
//...依此類推，添加共 4 個Stroke objects
```



Serialization 實例

: 備妥 collection 並添加 Rect, Circle 元素

➤ class Rect, class Circle 稍後說明

```
Rect r = new Rect(new Integer(0x11),  
                  new Integer(0x22),  
                  new Integer(0x33),  
                  new Integer(0x44));  
Circle c = new Circle(new Integer(0x55),  
                      new Integer(0x66),  
                      new Integer(0x77));
```

➤ reference semantics

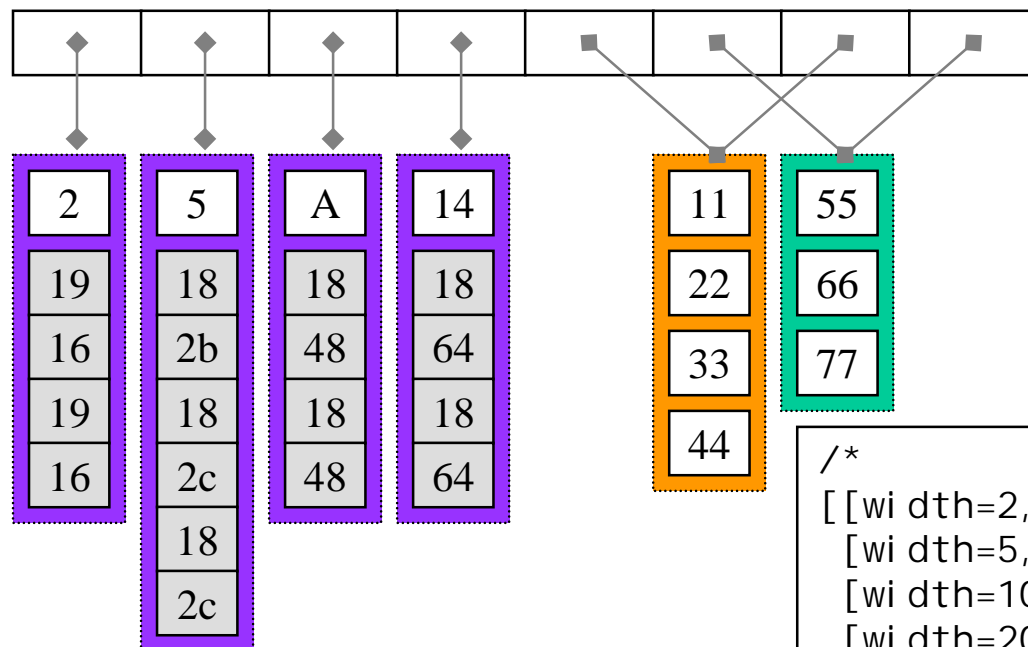
```
sl.add(r);    sl.add(c);  
sl.add(r);    sl.add(c);  
System.out.println(sl); //結果見下頁
```



Serialization 實例

: 元素備妥後的狀態

LinkedList<Object> sl;



➤ System.out.println(sl) 的輸出結果

```
/*
[[width=2, points=[25, 22, 25, 22]],
 [width=5, points=[24, 43, 24, 44, 24, 44]],
 [width=10, points=[24, 72, 24, 72]],
 [width=20, points=[24, 100, 24, 100]],
 [L=17, T=34, W=51, H=68], [X=85, Y=102, R=119],
 [L=17, T=34, W=51, H=68], [X=85, Y=102, R=119]]
*/
```



Serialization 實例

: class Stroke

- class Stroke 實現 Serializable，所以可以 serialized to stream。
- class Stroke 實作有 toString()，所以可以配合 System.out.println()。

```
public class Stroke implements Serializable
{
    Integer m_width;
    ArrayList<Integer> m_i a;

    ... //ctor()

    public String toString() {
        return "[width=" + m_width +
            ", points=" + m_i a + " ]";
    }
}
```



Serialization 實例

: class Rect

- class Rect 實現 Serializable，所以可以 serialized to stream。
- class Rect 實作有 toString()，所以可以配合 System.out.println()。

```
public class Rect implements Serializable
{
    Integer m_left, m_top, m_width, m_height;

    ... //ctor(), getHei ght(), setHei ght()

    public String toString() {
        return "[L=" + m_left + ", T=" + m_top +
            ", W=" + m_width + ", H=" + m_height + " ]";
    }
}
```




Serialization 實例

: class Circle

- class Circle 實現 Serializable，所以可以 serialized to stream。
- class Circle 實作有 toString()，所以可以配合 System.out.println()。

```
public class Circle implements Serializable
{
    Integer m_x, m_y, m_r;

    ...//ctor()

    public String toString() {
        return "[X=" + m_x + ", Y=" + m_y + ", R=" + m_r + "];"
    }
}
```



Serialization 實例

: header of LinkedList and ArrayList

- class LinkedList 實現 Serializable，所以可以 serialized to stream。
- class ArrayList 實現 Serializable，所以可以 serialized to stream。

```
public class LinkedList extends AbstractSequentialList
    implements List,
        Cloneable,
        java.io.Serializable;

public class ArrayList extends AbstractList
    implements List,
        RandomAccess,
        Cloneable,
        java.io.Serializable;
```

- 這兩個 Collection Classes 除宣告實現 Serializable 外，內部還動用了其他 Serialization 設施，如 transient, readObject(), writeObject()。



Serialization 實例

: LinkedList

➤ 後面另有投影片談到LinkedList 中的fields

➤ class LinkedList 實作碼中有 writeObject()，所以 serialized 時被喚起，製作出不同於 default serialized form 之 custom serialized form.

ObjectOutputStream.writeObject



writeObject0



writeOrdinaryObject



writeSerialData



ObjectStreamClass slotDesc
slotDesc.invokeWriteObject



```
private synchronized void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException {
    // Write out any hidden serialization magic
    s.defaultWriteObject();

    // Write out size
    s.writeInt(size);

    // Write out all elements in the proper order.
    for (Entry e = header.next; e != header; e = e.next)
        s.writeObject(e.element);
}
```



Serialization 實例

: ArrayList

➤ 後面另有投影片談到ArrayList 中的fields

➤ class ArrayList 實作碼中有 writeObject() ,
所以 serialized 時被喚起，製作出不同於 default serialized form 之
custom serialized form.

ObjectOutputStream.writeObject

↳ writeObject0

↳ writeOrdinaryObject

↳ writeSerialData

↳ ObjectOutputStream class slotDesc
slotDesc. invokeWriteObject



```
private synchronized void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException{
    // Write out element count, and any hidden stuff
    s.defaultWriteObject();

    // Write out array length
    s.writeInt(elementData.length);

    // Write out all elements in the proper order.
    for (int i=0; i<size; i++)
        s.writeObject(elementData[i]);
}
```



Serialization 實例

: Integer & Number

- class Number 實現 Serializable，所以可以 serialized to stream。
- class Integer 繼承 Number，所以也可以 serialized to stream。
- serialVersionUID (Unique Identifier) 後述。

➤ 如果未曾「宣告一個名為 serialVersionUID 的 private static final long 欄位」，系統會以一個繁複程序自動為你的 class 生成一個，其值受到 class 名稱、class 所實現之 interfaces 名稱、class 的所有 public 和 protected 成員名稱的影響。選擇什麼樣的值影響並不大。常見做法是針對你的 class 執行 **serialver** 便可獲得一個可用數值。任意編造一個值也是可以的。

```
public final class Integer extends Number implements Comparable {
    ...
    /**
     * The value of the <code>Integer</code>.
     * @serial
     */
    private int value;
}
```

```
public abstract class Number implements java.io.Serializable {
    ...
    /** use serialVersionUID from JDK 1.0.2 for interoperability */
    private static final long serialVersionUID = -8742448824652078965L;
}
```

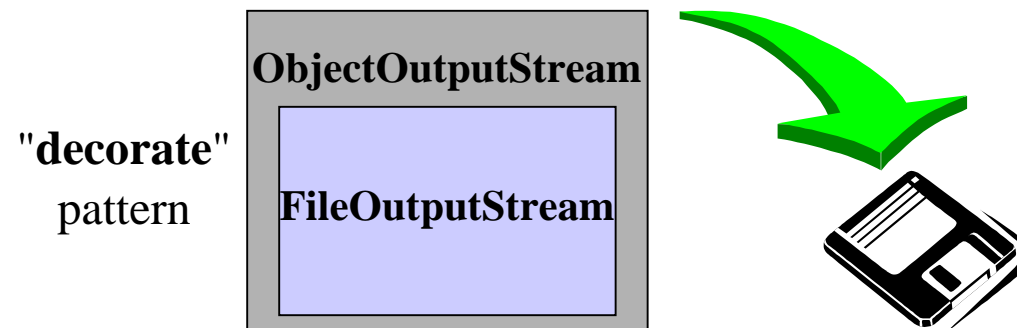


Serialization 實例

: 寫至檔案

- 通常objects 以 `writeObject()` 寫出
- primitives 則以 `DataOutput's methods` (例如 `WriteInt()`, `WriteFloat()`...) 寫出

`ObjectOutputStream.writeObject(xx)`



```
ObjectOutputStream out =
    new ObjectOutputStream(
        new FileOutputStream("collect.out"));

out.writeObject(sl);
out.close(); // also flush output stream
```



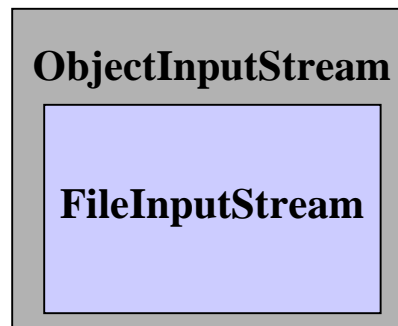
Serialization 實例

: 讀自檔案 (deSerialization)

- 通常 objects 以 readObject() 讀入
- primitives 則以 DataInput's methods (例如 ReadInt(), ReadFloat()...) 讀入

`obj = ObjectInputStream.readObject();`

"decorate"
pattern



```
ObjectInputStream in =
    new ObjectInputStream(
        new FileInputStream("collect.out"));
```

```
LinkedList sl2 = (LinkedList) in.readObject();
in.close();
System.out.println(sl2); // 結果見下頁
```

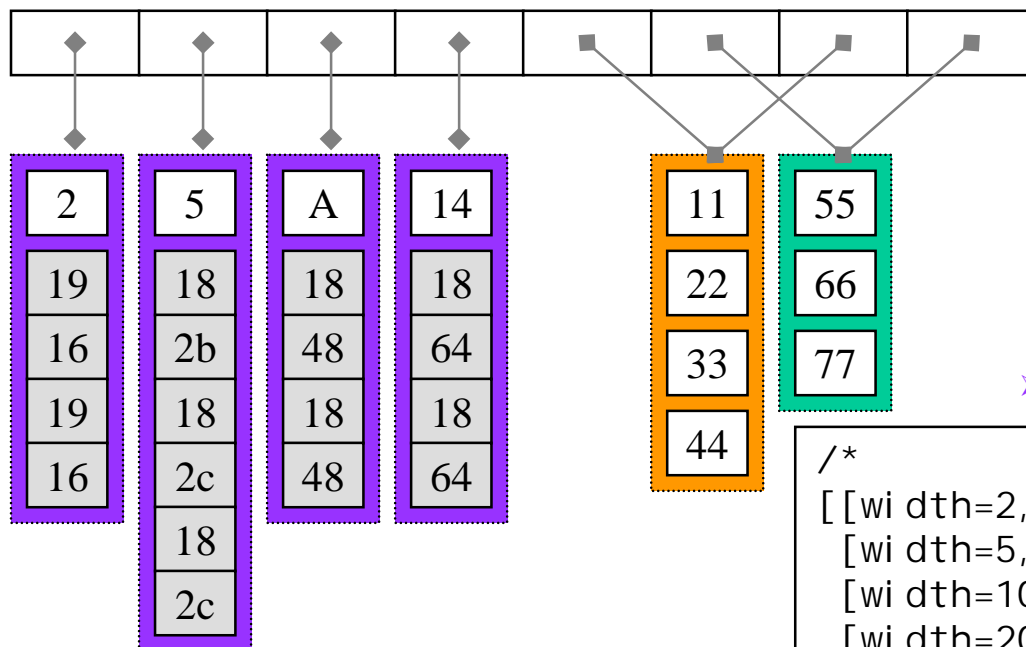
➤ 讀出來是Object，
需自行轉型



Serialization 實例

: 從檔案讀出後比對

LinkedList<Object> s12;



➤ System.out.println(s12) 的輸出結果。與 s1 完全相同

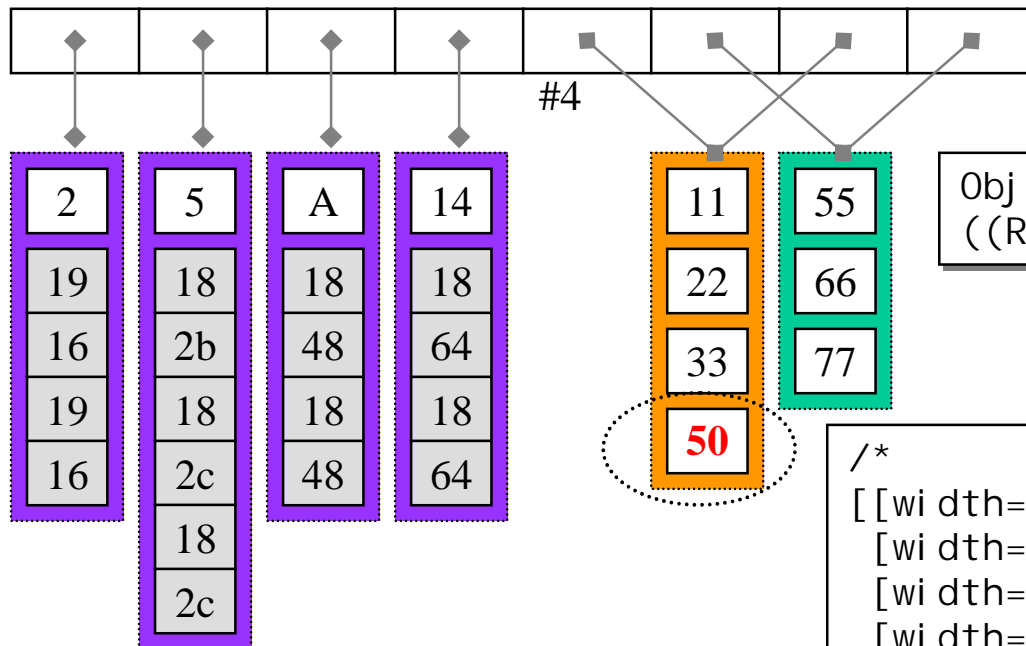
```
/*
[[width=2, points=[25, 22, 25, 22]],
 [width=5, points=[24, 43, 24, 44, 24, 44]],
 [width=10, points=[24, 72, 24, 72]],
 [width=20, points=[24, 100, 24, 100]],
 [L=17, T=34, W=51, H=68], [X=85, Y=102, R=119],
 [L=17, T=34, W=51, H=68], [X=85, Y=102, R=119]]
*/
```



Serialization 實例

: 驗證 reference semantics

- 修改 #4 元素內容，會連帶影響 #6 元素內容。證明是Java Collections是"reference semantics"
- 原容器如此，deSerialization後的新容器也如此，證明原結構（或說原狀態、原本的graph/web）被忠實回復。



- 修改 #4 元素（一個Rect object）的 height

```
Object o = sl.get(4);
((Rect)o).setHeight(new Integer(0x50));
```

- sl 經過修改後，println 的輸出結果。sl2 亦同。

```
/*
[[width=2, points=[25, 22, 25, 22]],
 [width=5, points=[24, 43, 24, 44, 24, 44]],
 [width=10, points=[24, 72, 24, 72]],
 [width=20, points=[24, 100, 24, 100]],
 [L=17, T=34, W=51, H=80], [X=85, Y=102, R=119],
 [L=17, T=34, W=51, H=80], [X=85, Y=102, R=119]]
*/
```



Serialization 結果

: 檔案傾印 (file dump)

運用任何 **binary dump** 工具程式，可觀察如下結果：

➤ 完整內容於下兩頁列出

```
Turbo Dump Version 5.0.16.12 Copyright (c) 1988, 2000 Inprise Corporation
Display of File COLLECT.OUT

000000: AC ED 00 05 73 72 00 14 6A 61 76 61 2E 75 74 69 秒..sr..java.util_
000010: 6C 2E 4C 69 6E 6B 65 64 4C 69 73 74 0C 29 53 5D l.LinkedList.)S]
000020: 4A 60 88 22 03 00 00 78 70 77 04 00 00 08 73 J`. "...xpw.....s
000030: 72 00 06 53 74 72 6F 6B 65 F2 D8 79 18 90 CC 00 r..Stroke懣y. .
000040: C1 02 00 02 4C 00 04 6D 5F 69 61 74 00 15 4C 6A ....L..m_iat..Lj
000050: 61 76 61 2F 75 74 69 6C 2F 41 72 72 61 79 4C 69 ava/util/ArrayLi
000060: 73 74 3B 4C 00 07 6D 5F 77 69 64 74 68 74 00 13 st;L..m_widthht..
000070: 4C 6A 61 76 61 2F 6C 61 6E 67 2F 49 6E 74 65 67 Ljava/lang/Integ
000080: 65 72 3B 78 70 73 72 00 13 6A 61 76 61 2E 75 74 er;xpsr..java.ut
000090: 69 6C 2E 41 72 72 61 79 4C 69 73 74 78 81 D2 1D il.ArrayListx .
0000A0: 99 C7 61 9D 03 00 01 49 00 04 73 69 7A 65 78 70 a....I..sizexp
0000B0: 00 00 00 04 77 04 00 00 00 0A 73 72 00 11 6A 61 ....w.....sr..ja
0000C0: 76 61 2E 6C 61 6E 67 2E 49 6E 74 65 67 65 72 12 va.lang.Integer.
0000D0: E2 A0 A4 F7 81 87 38 02 00 01 49 00 05 76 61 6C ?父?8...I..val
```



Serialization 結果

serialized file 完整傾印, 1/2

Turbo Dump Version 5.0.16.12 Copyright (c) 1988, 2000 Inprise Corporation
Display of File COLLECT.OUT

```

000000: AC ED 00 05 73 72 00 14 6A 61 76 61 2E 75 74 69 秒..sr..java.uti_
000010: 6C 2E 4C 69 6E 6B 65 64 4C 69 73 74 0C 29 53 5D l.LinkedList.)S]
000020: 4A 60 88 22 03 00 00 78 70 77 04 00 00 00 08 73 J`. "...xpw.....s
000030: 72 00 06 53 74 72 6F 6B 65 F2 D8 79 18 90 CC 00 r..Stroke 慰y. .
000040: C1 02 00 02 4C 00 04 6D 5F 69 61 74 00 15 4C 6A ....L..m_iat..Lj
000050: 61 76 61 2F 75 74 69 6C 2F 41 72 72 61 79 4C 69 ava/util/ArrayLi
000060: 73 74 3B 4C 00 07 6D 5F 77 69 64 74 68 74 00 13 st;L..m_widtht..
000070: 4C 6A 61 76 61 2F 6C 61 6E 67 2F 49 6E 74 65 67 Ljava/lang/Integ
000080: 65 72 3B 78 70 73 72 00 13 6A 61 76 61 2E 75 74 er;xpsr..java.ut
000090: 69 6C 2E 41 72 72 61 79 4C 69 73 74 78 81 D2 1D il.ArrayListx .
0000A0: 99 C7 61 9D 03 00 01 49 00 04 73 69 7A 65 78 70 a....I..sizexp
0000B0: 00 00 00 04 77 04 00 00 00 0A 73 72 00 11 6A 61 ....w.....sr..ja
0000C0: 76 61 2E 6C 61 6E 67 2E 49 6E 74 65 67 65 72 12 va.lang.Integer.
0000D0: E2 A0 A4 F7 81 87 38 02 00 01 49 00 05 76 61 6C ?父?8...I..val
0000E0: 75 65 78 72 00 10 6A 61 76 61 2E 6C 61 6E 67 2E uexr..java.lang.
0000F0: 4E 75 6D 62 65 72 86 AC 95 1D 0B 94 E0 8B 02 00 Number ... ..
000100: 00 78 70 00 00 00 19 73 71 00 7E 00 08 00 00 00 .xp....sq.~.....
000110: 16 73 71 00 7E 00 08 00 00 00 19 73 71 00 7E 00 .sq.~.....sq.~.
000120: 08 00 00 00 16 78 73 71 00 7E 00 08 00 00 00 02 .....xsq.~.....
000130: 73 71 00 7E 00 02 73 71 00 7E 00 06 00 00 00 06 sq.~..sq.~.....
000140: 77 04 00 00 00 0A 73 71 00 7E 00 08 00 00 00 18 w.....sq.~.....
000150: 73 71 00 7E 00 08 00 00 00 2B 73 71 00 7E 00 08 sq.~.....+sq.~..
000160: 00 00 00 18 73 71 00 7E 00 08 00 00 00 2C 73 71 ....sq.~.....,sq
000170: 00 7E 00 08 00 00 00 18 73 71 00 7E 00 08 00 00 .~.....sq.~.....
000180: 00 2C 78 73 71 00 7E 00 08 00 00 00 05 73 71 00 .,xsq.~.....sq.
000190: 7E 00 02 73 71 00 7E 00 06 00 00 00 04 77 04 00 ~..sq.~.....w..

```



Serialization 結果

: 檔案傾印 (file dump) 2/2

serialized file 完整傾印 (續上頁) 2/2

```

0001A0: 00 00 0A 73 71 00 7E 00 08 00 00 00 18 73 71 00 ...sq.~.....sq.
0001B0: 7E 00 08 00 00 00 48 73 71 00 7E 00 08 00 00 00 ~.....Hsq.~.....
0001C0: 18 73 71 00 7E 00 08 00 00 00 48 78 73 71 00 7E .sq.~.....Hxsq.~
0001D0: 00 08 00 00 00 0A 73 71 00 7E 00 02 73 71 00 7E .....sq.~..sq.~
0001E0: 00 06 00 00 00 04 77 04 00 00 00 0A 73 71 00 7E .....w.....sq.~
0001F0: 00 08 00 00 00 18 73 71 00 7E 00 08 00 00 00 64 .....sq.~.....d
000200: 73 71 00 7E 00 08 00 00 00 18 73 71 00 7E 00 08 sq.~.....sq.~..
000210: 00 00 00 64 78 73 71 00 7E 00 08 00 00 00 14 73 ...dxsq.~.....s
000220: 72 00 04 52 65 63 74 85 E1 2D 3E 3D A4 30 AA 02 r..Rect ->=.0..
000230: 00 04 4C 00 08 6D 5F 68 65 69 67 68 74 71 00 7E ..L..m_heightq.~
000240: 00 04 4C 00 06 6D 5F 6C 65 66 74 71 00 7E 00 04 ..L..m_leftq.~..
000250: 4C 00 05 6D 5F 74 6F 70 71 00 7E 00 04 4C 00 07 L..m_topq.~..L..
000260: 6D 5F 77 69 64 74 68 71 00 7E 00 04 78 70 73 71 m_widthq.~..xpsq
000270: 00 7E 00 08 00 00 00 44 73 71 00 7E 00 08 00 00 ~.....Dsq.~.....
000280: 00 11 73 71 00 7E 00 08 00 00 00 22 73 71 00 7E ..sq.~....."sq.~
000290: 00 08 00 00 00 33 73 72 00 06 43 69 72 63 6C 65 .....3sr..Circle
0002A0: C3 76 A9 B7 F7 E4 F7 FF 02 00 03 4C 00 03 6D 5F 瀚弧瀾.....L..m_
0002B0: 72 71 00 7E 00 04 4C 00 03 6D 5F 78 71 00 7E 00 rq.~..L..m_xq.~.
0002C0: 04 4C 00 03 6D 5F 79 71 00 7E 00 04 78 70 73 71 .L..m_yq.~..xpsq
0002D0: 00 7E 00 08 00 00 00 77 73 71 00 7E 00 08 00 00 ~.....wsq.~.....
0002E0: 00 55 73 71 00 7E 00 08 00 00 00 66 71 00 7E 00 .Usq.~.....fq.~.
0002F0: 27 71 00 7E 00 2D 78 00 00 00 00 00 00 00 00 00 'q.~.-x.....

```



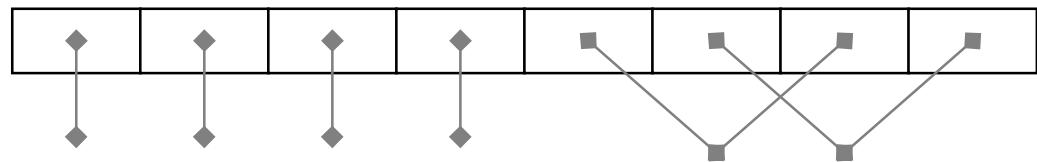
Serialization 結果剖析 (1)

: stream header, LinkedList's type & partial state

➤ 注意，Serialization 的處理方式是遞迴式地深入處理每一個 object（內的 referenced objects）遇有先前已記錄之 type or object，便只儲存 handle。

➤ 以下記錄LinkedList 的 type info 和 state

LinkedList sl;



```

AC ED          STREAM_MAGIC
00 05          STREAM_VERSION
73            TC_OBJECT
72            TC_CLASSDESC
00 14 6A 61 76 61 2E 75 74 69 6C 2E 4C 69 6E 6B 65 64 4C 69 73 74
    "java.util.LinkedList"
0C 29 53 5D 4A 60 88 22 03 00 00
    SerialVersionUID(long)+flag, 後有0個members需記錄
78            TC_ENDBLOCKDATA
70            TC_NULL
77            TC_BLOCKDATA
04            block length
00 00 00 08
    LinkedList共有8個元素（4個Strokes, 2個Rects, 2個Circles）
  
```

➤ non-transient fields (except SerialVersionUID)

➤ WriteObject()寫出的 fields



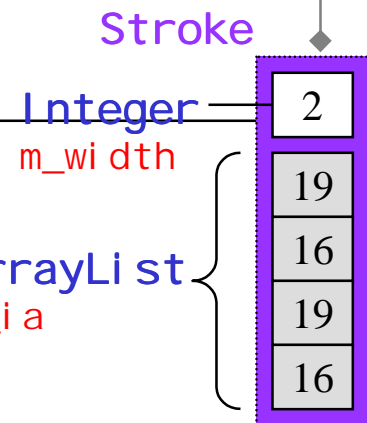
Serialization 結果剖析 (2)

: 1st element (Stroke's) type

➤ Stroke class 內有兩個 fields；以下記錄 field's name & type。

```

73          TC_OBJECT
72          TC_CLASSDESC
00 06 53 74 72 6F 6B 65          "Stroke"
F2 D8 79 18 90 CC 00 C1 02 00 02
    SerialVersionUID(long)+flag, 後有2個members需記錄
4C          'L' (class or interface)
00 04 6D 5F 69 61          "m_ia"
74          TC_STRING
00 15 4C 6A 61 76 61 2F 75 74 69 6C 2F 41 72 72 61 79 4C 69 73 74 3B
    "Ljava/util/ArrayList;"
4C          'L' (class or interface)
00 07 6D 5F 77 69 64 74 68 "m_width"
74          TC_STRING
00 13 4C 6A 61 76 61 2F 6C 61 6E 67 2F 49 6E 74 65 67 65 72 3B
    "Ljava/lang/Integer;"
78          TC_ENDBLOCKDATA
70          TC_NULL
  
```

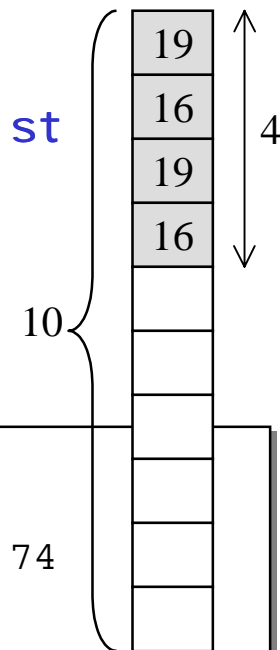




Serialization 結果剖析 (3)

: ArrayList's type & partial state

ArrayLi st



➤ Stroke class 內有兩個 fields。以下記錄第一個field (ArrayList) 的 type info 和 state。

```
73          TC_OBJECT
72          TC_CLASSDESC
00 13 6A 61 76 61 2E 75 74 69 6C 2E 41 72 72 61 79 4C 69 73 74
   "java.util.ArrayList"
```

```
78 81 D2 1D 99 C7 61 9D 03 00 01
```

SerialVersionUID(long)+flag, 後有1個members需記錄

```
49          'I'(int)
```

```
00 04 73 69 7A 65   "size"
```

➤ non-transient fields
(except SerialVersionUID)

```
78          TC_ENDBLOCKDATA
```

```
70          TC_NULL
```

```
00 00 00 04       ArrayList 共有4個元素
```

```
77          TC_BLOCKDATA
```

```
04          block length
```

```
00 00 00 0A       ArrayList array length
```

➤ WriteObject()寫出的 fields



Serialization 結果剖析 (4)

: 1st element (Integer's) type & state

➤ ArrayList 內有四個同型元素。以下記錄元素型別 (Integer) 的 type info。

Number
↑
Integer

19
16
19
16
~~~~~

```

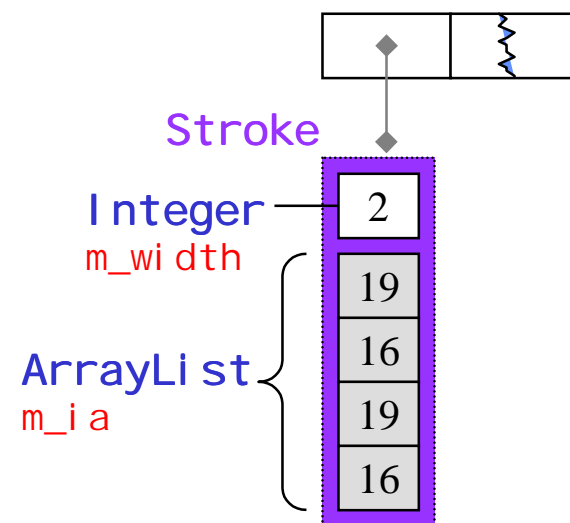
73          TC_OBJECT
72          TC_CLASSDESC
00 11 6A 61 76 61 2E 6C 61 6E 67 2E 49 6E 74 65 67 65 72
    "java.lang.Integer"
12 E2 A0 A4 F7 81 87 38 02 00 01
    SerialVersionUID(long)+flag, 後有1個members需記錄
49          'I' (int)
00 05 76 61 6C 75 65          "value"
78          TC_ENDBLOCKDATA
72          TC_CLASSDESC
00 10 6A 61 76 61 2E 6C 61 6E 67 2E 4E 75 6D 62 65 72
    "java.lang.Number"
86 AC 95 1D 0B 94 E0 8B 02 00 00
    SerialVersionUID(long)+flag, 後有0個members需記錄
78          TC_ENDBLOCKDATA
70          TC_BASE (TC_NULL)
00 00 00 19          array 第一個值(25)
  
```



# Serialization 結果剖析 (5)

: Stroke's remnant state

73 71	TC_OBJECT, TC_REFERENCE
00 7E 00 08	array 第二個值(22)
73 71	TC_OBJECT, TC_REFERENCE
00 7E 00 08	array 第三個值(25)
73 71	TC_OBJECT, TC_REFERENCE
00 7E 00 08	array 第四個值(22)
78	TC_ENDBLOCKDATA
73 71	TC_OBJECT, TC_REFERENCE
00 7E 00 08	筆寬
00 00 00 02	



面對 Integer，由於先前已經登錄過，所以這兒使用 reference 即可。8 是 handle。



# Serialization 結果剖析 (6)

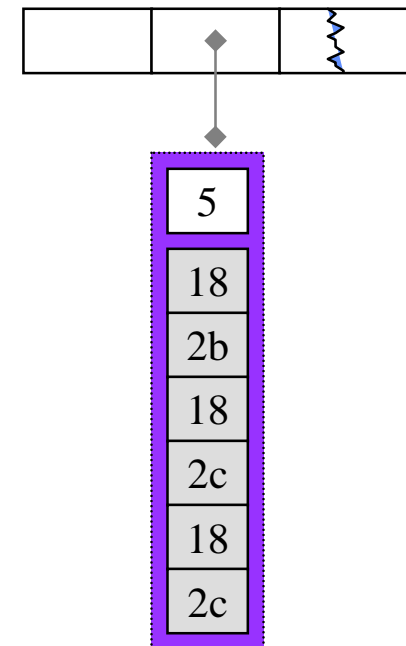
## : 2nd element's type

- 由於LinkedList 可存放任何 Object，所以每一個元素都必須記錄其實際type。

面對 **Stroke**，由於先前已經登錄過，所以這兒使用 reference 即可。2是handle。

73	71					TC_OBJECT, TC_REFERENCE
00	7E	00	02			
73	71					TC_OBJECT, TC_REFERENCE
00	7E	00	06			

面對 **ArrayList**，由於先前已經登錄過，所以這兒使用 reference 即可。6是handle。



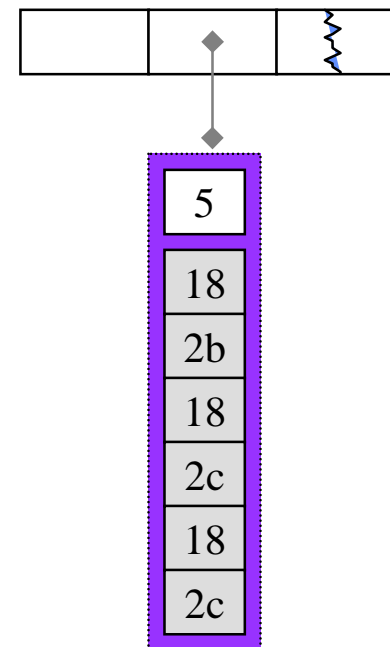


# Serialization 結果剖析 (7)

: 2nd element's state

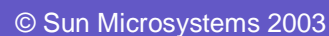
00 00 00 06	ArrayList 共有6個元素
77 04	TC_BLOCKDATA+length
00 00 00 0A	ArrayList array length
73 71	TC_OBJECT, TC_REFERENCE
00 7E 00 08	
00 00 00 18	array 第一個值(24)
73 71	
00 7E 00 08	
00 00 00 2B	array 第二個值(43)
73 71	
00 7E 00 08	
00 00 00 18	array 第三個值(24)
73 71	
00 7E 00 08	
00 00 00 2C	array 第四個值(44)
73 71	
00 7E 00 08	
00 00 00 18	array 第五個值(24)
73 71	
00 7E 00 08	
00 00 00 2C	array 第六個值(44)
78	TC_ENDBLOCKDATA
73 71	
00 7E 00 08	
00 00 00 05	筆寬

面對 **Integer**，由於先前已經登錄過，所以這兒使用 reference 即可。8是handle。





- 面對 **Stroke**，由於先前已經登錄過，所以這兒使用 reference 即可。2是handle。

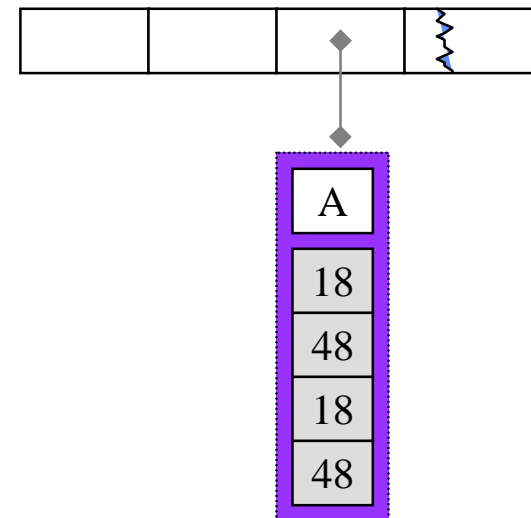




# Serialization 結果剖析 (9)

: 3rd element's state



00 00 00 04	ArrayList 共有4個元素
77 04	TC_BLOCKDATA+length
00 00 00 0A	ArrayList array length
73 71	TC_OBJECT, TC_REFERENCE
00 7E 00 08	
00 00 00 18	array 第一個值(24)
73 71	
00 7E 00 08	
00 00 00 48	array 第二個值(72)
73 71	
00 7E 00 08	
00 00 00 18	array 第三個值(24)
73 71	
00 7E 00 08	
00 00 00 48	array 第四個值(72)
78	TC_ENDBLOCKDATA
73 71	
00 7E 00 08	
00 00 00 0A	筆寬



面對 **Integer**，由於先前已經登錄過，所以這兒使用 reference 即可。8是handle。





- |  |  |  |                                                                                     |                                                                                     |
|--|--|--|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|  |  |  |  |  |
|--|--|--|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|

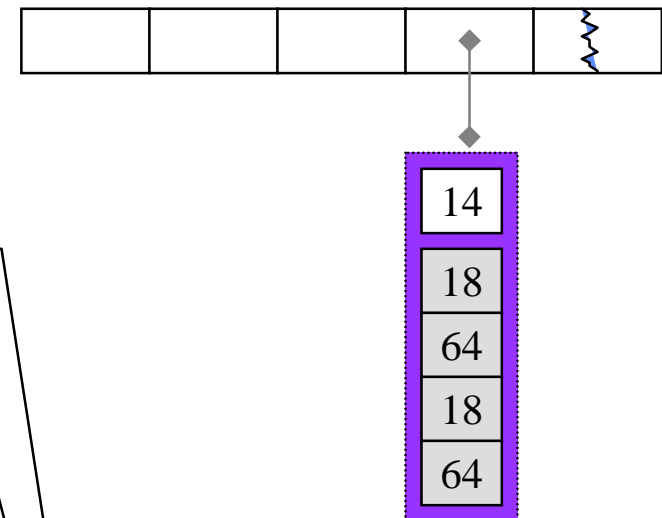
14
18
64
18
64



# Serialization 結果剖析 (11)

: 4th element's state

00 00 00 04	ArrayList 共有4個元素
77 04	TC_BLOCKDATA+length
00 00 00 0A	ArrayList array length
73 71	TC_OBJECT, TC_REFERENCE
00 7E 00 08	
00 00 00 18	array 第一個值(24)
73 71	
00 7E 00 08	
00 00 00 64	array 第二個值(100)
73 71	
00 7E 00 08	
00 00 00 18	array 第三個值(24)
73 71	
00 7E 00 08	
00 00 00 64	array 第四個值(100)
78	TC_ENDBLOCKDATA
73 71	
00 7E 00 08	
00 00 00 14	筆寬



面對 **Integer**，由於先前已經登錄過，所以這兒使用 reference 即可。8是handle。



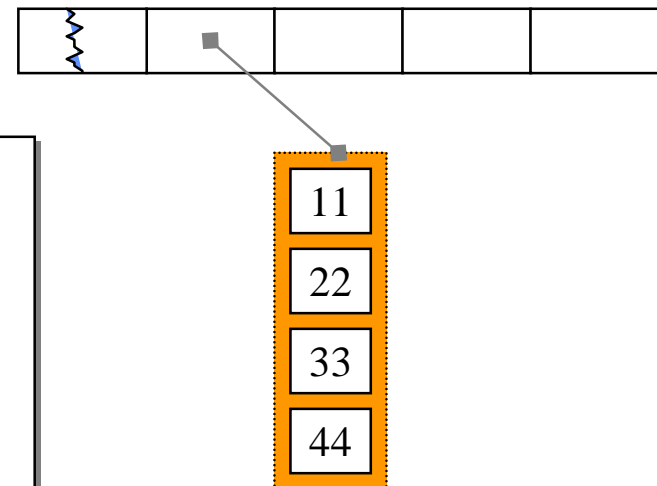
# Serialization 結果剖析 (12)

## : 5th element's type

➤ 記錄Rect的type info。

```

73          TC_OBJECT
72          TC_CLASSDESC
00 04 52 65 63 74          "Rect "
85 E1 2D 3E 3D A4 30 AA 02 00 04
    SerialVersionUID(long)+flag, 後有4個members需記錄
4C
00 08 6D 5F 68 65 69 67 68 74          "m_height "
71 00 7E 00 04 4C
00 06 6D 5F 6C 65 66 74          "m_left "
71 00 7E 00 04 4C
00 05 6D 5F 74 6F 70          "m_top "
71 00 7E 00 04 4C
00 07 6D 5F 77 69 64 74 68          "m_width "
71 00 7E 00 04
78          TC_ENDBLOCKDATA
70
  
```



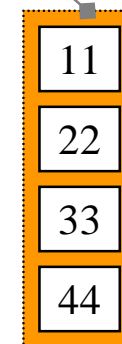


# Serialization 結果剖析 (13)

: 5th element's state

➤ 記錄Rect的field data。

73 71	TC_OBJECT, TC_REFERENCE
00 7E 00 08	m_high 的值
00 00 00 44	
73 71	
00 7E 00 08	m_left 的值
00 00 00 11	
73 71	
00 7E 00 08	m_top 的值
00 00 00 22	
73 71	
00 7E 00 08	m_width 的值
00 00 00 33	



面對 **Integer**，由於先前已經登錄過，所以這兒使用 reference 即可。8是handle。



# Serialization 結果剖析 (14)

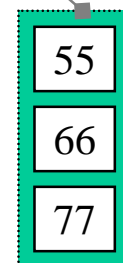
: 6th element's type



➤ 記錄Circle的type info。

```

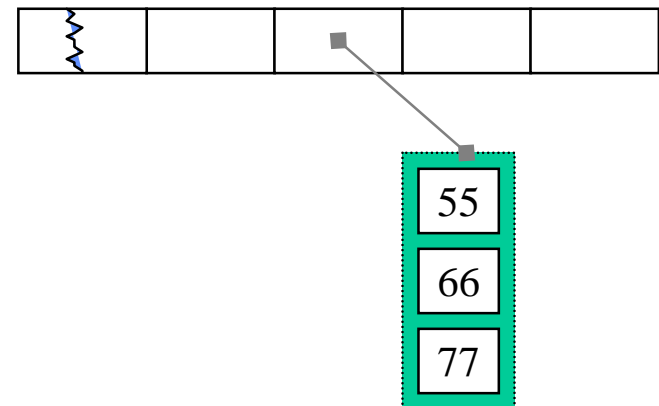
73          TC_OBJECT
72          TC_CLASSDESC
00 06 43 69 72 63 6C 65    "Circle"
C3 76 A9 B7 F7 E4 F7 FF 02 00 03
    SerialVersionUID(long)+flag, 後有3個members需記錄
4C
00 03 6D 5F 72            "m_r"
71 00 7E 00 04 4C
00 03 6D 5F 78            "m_x"
71 00 7E 00 04 4C
00 03 6D 5F 79            "m_y"
71 00 7E 00 04
78          TC_ENDBLOCKDATA
70
  
```





# Serialization 結果剖析 (15)

: 6th element's state



➤ 記錄Circle的field data。

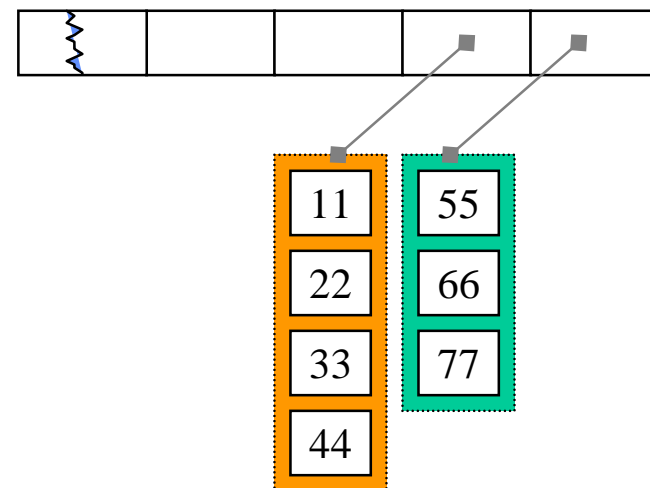
73 71	TC_OBJECT, TC_REFERENCE
00 7E 00 08	m_r 的值
00 00 00 77	
73 71	
00 7E 00 08	m_x 的值
00 00 00 55	
73 71	
00 7E 00 08	m_y 的值
00 00 00 66	

面對 **Integer**，由於先前已經登錄過，所以這兒使用 reference 即可。8是handle。



# Serialization 結果剖析 (16)

: 7th element & 8th element



71	TC_REFERENCE
00 7E 00 27	
71	TC_REFERENCE
00 7E 00 2D	
78	TC_ENDBLOCKDATA

先前已經登錄過完全相同的  
**Rect object**和**Circle object**，  
所以這兒使用 reference 即可



## Serialization 結果剖析 (17)

### : 總結

Serialization 的處理方式是遞迴式地深入處理每一個 object（內的 referenced objects）；遇有先前已記錄之 type or object，便只儲存 handle。

以本例而言，`ObjectOutputStream.writeObject()` 面對 object，首先將其 class info (`LinkedList`) 及 serializable state 寫出，而後面對其第一個元素，首先將其 class info (`Stroke`) 寫出，而後面對其第一個成分，首先將其 class info (`ArrayList`) 及 serializable state 寫出，而後面對其第一個元素，首先將其 class info (`Integer`) 寫出，再寫出其 value；然後再寫出 `ArrayList` 的第 2,3,4 個元素，待 `ArrayList` 處理完畢後再回頭處理 `Stroke` 的第二個成分 (`Integer`)。 `Stroke` 處理完畢後再回頭處理 `LinkedList` 的第 2,3,4 個元素。然後是第 5 個元素 `Rect` object，然後是第 6 個元素 `Circle` object。然後是第 7 個元素 `Rect` object，然後是第 8 個元素 `Circle` object。





# deSerialization

## : 內部技術概觀 (1)

除了 **Class**, **ObjectStreamClass**, **String**, **array** 之外，對於其他任何 objects，都應從 stream 讀其 **ObjectStreamClass** 然後重新獲得 (retrieved) 其 local class，然後配置 (allocated) 一個 instance 並將它和它的 handle 放到 the set of known objects，再適當地回復其內容。



## deSerialization

### :內部技術概觀 (2)

對於 serializable objects，其第一個 non-serializable supertype 的 no-arg ctor 會被 (Serialization 機制) 喚起，每個 fields 獲得預設值。然後 class-specific readObject() (如未有定義則淪為 defaultReadObject()) 被喚起，以回復 fields 內容。注意，field 的 initializers 和 constructors 並不會獲得執行。



# Class Descriptor

: java.io.ObjectStreamClass

ObjectStreamClass 負責提供 "儲存於 stream 內的 class" 的資訊。stream內提供的是 class 的 fully-qualified name 及 serialization version UID (用來鑑定/驗明unique original class version) 。

```
package java.io;
public class ObjectStreamClass
{
    public static ObjectStreamClass lookup(Class cl);
    public String getName();
    public Class forClass();
    public ObjectStreamField[] getFields();
    public long getSerialVersionUID();
    public String toString();
}
```



# Class Descriptor

: 內含許多與 Reflection 機制相關的欄位

➤ 有了 class name，就可以取其 **Class** object，而後就可以取其 **Class Descriptor**，而後就可以利用 **Reflection** 取得該class的所有信息（有哪些fields，哪些methods，哪些 ctors...，哪些特定的 fields 和 methods...）

```
private static final ObjectStreamField[] serialPersistentFields;  
private Class cl;  
private ObjectStreamField[] fields; （運用getSerialFields()獲得，見次頁）  
private Constructor cons; （運用Class.getDeclaredConstructor()獲得）  
private Method writeObjectMethod; （運用Class.getDeclaredMethod()獲得）  
private Method readObjectMethod; （同上）  
private Method readObjectNoDataMethod; （同上）  
private Method writeReplaceMethod; （同上）  
private Method readResolveMethod; （同上）  
private ObjectStreamClass localDesc;  
private ObjectStreamClass superDesc;
```



# Class Descriptor

:內含許多與 Reflection 機制相關的函式

```
private static ObjectOutputStreamField[] getSerialFields(Class cl) {  
    ObjectOutputStreamField[] fields;  
    if (...)  
    {  
        if ((fields = getDeclaredSerialFields(cl)) == null) {  
            fields = getDefaultSerialFields(cl);  
        }  
        Arrays.sort(fields);  
    } else {  
        fields = NO_FIELDS;  
    }  
    return fields;  
}
```

次頁

次次頁



# Class Descriptor

: 內含許多與 Reflection 機制相關的函式

- 利用Reflection找出class中以 serialPersistentFields宣告的欄位（下5頁有例），一一記錄於ObjectStreamField[] 後傳!

```
private static ObjectStreamField[] getDeclaredSerialFields(Class cl) {
    ObjectStreamField[] serialPersistentFields = null;
    Field f = cl.getDeclaredField("serialPersistentFields");
    ...
    ObjectStreamField[] boundFields =
        new ObjectStreamField[serialPersistentFields.length];
    for (int i = 0; i < serialPersistentFields.length; i++) {
        ObjectStreamField spf = serialPersistentFields[i];
        Field f = cl.getDeclaredField(spf.getName());
        if ((f.getType() == spf.getType()) &&
            ((f.getModifiers() & Modifier.STATIC) == 0)) {
            boundFields[i] =
                new ObjectStreamField(f, spf.isUnshared(), true);
        }
        ...
    }
    return boundFields;
}
```

次次頁



# Class Descriptor

:內含許多與 Reflection 機制相關的函式

- 利用Reflection，找出class中的 non-static, non-transient 欄位，一一記錄於ObjectStreamField[] 內傳出。

```
private static ObjectStreamField[] getDefaultSerialFields(Class cl) {  
    Field[] clFields = cl.getDeclaredFields();  
    ArrayList list = new ArrayList();  
    int mask = Modifier.STATIC | Modifier.TRANSIENT;  
  
    for (int i = 0; i < clFields.length; i++) {  
        if ((clFields[i].getModifiers() & mask) == 0) {  
            list.add(new ObjectStreamField(clFields[i], false, true));  
        }  
    }  
    int size = list.size();  
    return (size == 0) ? NO_FIELDS :  
        (ObjectStreamField[]) list.toArray(new ObjectStreamField[size]);  
}
```





# ObjectStreamField

:內含許多與 Reflection 機制相關的欄位

```
/** field name */  
private final String name;  
/** canonical JVM signature of field type */  
private final String signature;  
/** field type (Object.class if unknown non-primitive type) */  
private final Class type;  
/** whether or not to (de)serialize field values as unshared */  
private final boolean unshared;  
/** corresponding reflective field object, if any */  
private final Field field;  
/** offset of field value in enclosing field group */  
private int offset = 0;
```





# deSerialization 結果剖析

## ：總結

- **Java deSerialization** 完全以 dynamic type system 為基礎，所以完全不需埋設暗樁函式。只要從 stream 中取得 class name，就可獲得其 **Class** object，進而獲得其 constructors, methods, serializable fields...。
- **C++** 屬於 static types system 語言，所以 **C++ serialization** 必須埋設暗樁函式（"**new xxx;**"）。埋設手法之優劣關係到程式庫的技術表現；通常使用 macros 完成任務。
- **dynamic type system**：程式用到的 classes，不必在編譯期出現，只需於執行期存在。
- **static type system**：程式用到的 classes，必須於編譯期出現。



# Serialization 編程注意事項

:定義 serializable class

- 必須實現 **java.io.Serializable** (package java.io)
- 必須確定哪些欄位應當被 **serializable** (可使用關鍵字 **transient**)
- 第一個 **non-serializable superclass** 的 **no-arg constructor** 會被喚起。
- 選擇性地定義 **writeObject()**, **readObject()**, **writeReplace()**, **readResolve()**。如果 class 提供了 **writeObject()** 和 **readObject()**，可在其中透過 **defaultWriteObject()** 和 **defaultReadObject()** 喚起 default serialization。因此一旦 class 實作兩函式，就有機會在 serializable field values 被寫前或被讀後加以修改。



# Serialization 編程注意事項

## : 定義 serializable fields

● **Default Serializable Fields:** 只要 **non-transient, non-static** 欄位都是。

● 運用 **serialPersistentFields** 改變上述  
(預設) 性質，例如：

- 經查驗，並未在 java.util 中發現任何這種用法
- 以下作法等同於 default serializable fields

```
class List implements Serializable {  
    List next;  
    private static final ObjectOutputStreamField[]  
        serialPersistentFields  
        = {new ObjectOutputStreamField("next", List.class)};  
}
```

↑  
name

↑  
type



# Serialization 編程注意事項

:爲 Serializable fields 提供文件 (Documenting)

- **@serial**: 針對每一個 default serializable field
- **@serialField**: 針對 serialPersistentFields (上頁) array中的每一個 ObjectOutputStreamField 成分。
- **@serialData**: 用以描述 the sequences and types of data written or read.



# Serialization 編程注意事項

: serializable fields , 以 java.util.LinkedList 為例

```
private transient Entry header = new Entry(null);
private transient int size = 0;

/**
 * @serialData The size of the list (the number of elements it
 *              contains) is emitted (int), followed by all of its
 *              elements (each an Object) in the proper order.
 */
private synchronized void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException {
    // Write out any hidden serialization magic
    s.defaultWriteObject();

    // Write out size
    s.writeInt(size);

    // Write out all elements in the proper order
    for (Entry e = header.next; e != null; e = e.next)
        s.writeObject(e.element);
}
```

```
private static class Entry {
    Object element;
    Entry next;
    Entry previous;
    ...
}
```

```
public class LinkedList
    extends AbstractSequentialList
    implements List,
        Cloneable,
        java.io.Serializable;
```



# Serialization 編程注意事項

: serializable fields , 以 java.util.ArrayList 為例

```
private static final Long serialVersionUID = 8683452581122892189L;
private transient Object elementData[];
/**
 * The size of the ArrayList (the number of elements it contains).
 * @serial
 */
private int size;

/**
 * @serial Data The length of the array backing the <tt>ArrayList</tt>
 * instance is emitted (int), followed by all of its elements
 * (each an <tt>Object</tt>) in the proper order.
 */
private synchronized void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException{
    // Write out element count, and any hidden stuff
    s.defaultWriteObject();

    // Write out array length
    s.writeInt(elementData.length);

    // Write out all elements in the proper order
    for (int i=0; i<size; i++)
        s.writeObject(elementData[i]);
}
```

```
public class ArrayList
    extends AbstractList
    implements List,
        RandomAccess,
        Cloneable,
        java.io.Serializable;
```



# deSerialization 編程注意事項

## :readObject(), 以 LinkedList 爲例

```
/**
 * Reconstitute this <tt>LinkedList</tt> instance from a stream
 * (that is deserialize it).
 */
private synchronized void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    // Read in any hidden serialization magic
    s.defaultReadObject();

    // Read in size
    int size = s.readInt();

    // Initialize header
    header = new Entry(null, null, null);
    header.next = header.previous = header;

    // Read in all elements in the proper order.
    for (int i=0; i<size; i++)
        add(s.readObject());
}
```

➤ 在 add() 中才建立 linkedlist 的關係，  
並非從 stream 讀入時就建立。這對 API 而言比較有彈性。





# deSerialization 編程注意事項

: readObject(), 以 ArrayList 為例

```
/**
 * Reconstitute the <tt>ArrayList</tt> instance from a stream
 * (that is, deserialize it).
 */
private synchronized void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    // Read in size, and any hidden stuff
    s.defaultReadObject();

    // Read in array length and allocate array
    int arrayLength = s.readInt();
    elementData = new Object[arrayLength];

    // Read in all elements in the proper order.
    for (int i=0; i<size; i++)
        elementData[i] = s.readObject();
}
```

➤ 在這裡才建立 array 的關係，並非從 stream 讀入時就建立。這使 API 較有彈性





## Serialization的代價

:ref. 《*Effective Java*》 item54

- 它會降低 class implementation 的變化彈性 — 在 class 發行 (released) 之後。
- 它會增加安全漏洞的可能性。
- 它會增加「發佈新版 class」時的測試負擔。

➤ 代價就是代價；代價不是缺點。該付出的代價就是要付。



# 何時不適用default Serialization?

:ref. 《Effective Java》 item55

- default serialized form是以該物件為根之物件圖（object graph）的物理表述（physical representation）的一個適度合理的高效編碼。它描述物件內含的資料，以及該物件所能觸及（reachable）的每一個其他物件。也描述所有這些彼此連結（interlinked）的物件所形成的拓樸關係（topology）。
- 理想的 serialized form只含物件的邏輯表述（logical representation），此與其物理表述（physical representation）彼此獨立。
- 如果物件的物理表述和其邏輯內容完全相同，那麼 default serialized form 對它而言是適當的



## default Serialization 適例

:ref. 《Effective Java》 item55, class Name

```
// Good candidate for default serialized form
public class Name implements Serializable {
    /**
     * Last name. Must be non-null.
     * @serial
     */
    private String lastName;

    /**
     * First name. Must be non-null.
     * @serial
     */
    private String firstName;

    /**
     * Middle initial, or '\u0000' if name lacks middle initial.
     * @serial
     */
    private char middleInitial;

    ... // Remainder omitted
}
```



## default Serialization 適例

:ref. 《*Effective Java*》 item55, class Name

- （續上頁）從邏輯角度看，姓名由代表名和姓的兩個字串及一個表示中間名大寫首字母的字元組成。class **Name**中的*instance* fields 明確反映出這些邏輯內容。
- 縱使default serialized form是合適的，你往往還是必須提供一個 **readObject()** 以確保 **invariants**（約束條件）和 **security**（安全防護）。以 **Name** 為例，**readObject()** 可確保 **lastName** 和 **firstName** 都是 non-null。

➤ invariants 和 security 各舉一例於後。



# default Serialization 不適例

:ref. 《*Effective Java*》 item55, StringList

➤ 一個由 String 形成的 list。

```
// Awful candidate for default serialized form
public class StringList implements Serializable {
    private int size = 0;
    private Entry head = null;

    private static class Entry implements Serializable {
        String data;
        Entry next;
        Entry previous;
    }

    ... // Remainder omitted
}
```



## default Serialization 不適例

:ref. 《*Effective Java*》 item55, StringList

- 邏輯上說，這個 class 表現出一系列string。物理上的表現則是個 doubly linked list。如果採用 default serialized form，將煞費苦心地鏡射出 linked list 的每一筆記錄（entry）以及各筆記錄之間的所有雙向連結（links）。
- 當物件的**物理表述**（physical representation）和其**邏輯資料**（logical data）有實質上的差別，如果你採用 default serialized form，會產生以下四個缺點：



## default Serialization 不適例

:ref. 《*Effective Java*》 item55, StringList

1. 永久性地將 exported API 繫縛 (ties) 於內部表述 (internal representation) 上。就本例而言，StringList 永遠無法擺脫「處理 linked lists」的程式碼 — 即使它不再使用 linked lists。
2. 可能消耗過多空間。entries 和 links 只是實作細目 (implementation details)，不值得含入 serialized form 內。serialized form 過大會造成寫入磁碟或傳輸於網絡時耗用過多時間。





## default Serialization 不適例

:ref. 《*Effective Java*》 item55, StringList

3.可能消耗過多時間。serialization logic並不知道本例的 topology of the object graph，因此必須經歷一場成本高昂的 graph traversal 行動。

4.可能造成stack滿溢（overflows）。預設的 serialization程序會對object graph執行遞迴走訪（recursive traversal），那可能會造成stack滿溢。造成 stack 滿溢的元素數量取決於JVM實作品——某些實作品或許完全沒有問題。





## default Serialization 不適例

:ref. 《*Effective Java*》 item55, hash table

hash table的物理表述是一系列hash buckets，內含一筆一筆的 key-value entries。一筆entry該被放到哪個bucket去呢？這與key的hash code有關，而hash code在不同的JVM實作品中並不保證相同，甚至同一個JVM實作品的每次運行都不保證有相同結果。因此一個hash table如果接受default serialized form，會造成嚴重錯誤。



## 像對待建構式一樣地對待readObject() :ref. 《Effective Java》 item56

- readObject()是另一個有效力的public建構式，你必須像對待任何建構式一樣地小心謹慎對待它。就像建構式必須檢查其引數的有效性並適當製作參數的保護性複本（defensive copies）一樣，readObject()也必須如此。如果沒能成功完成上述事情，就會導致攻擊者可以輕鬆違反 class 的約束條件（invariants）。
- 寬鬆地說，readObject() 是個建構式，接受一組 byte stream 做為唯一參數。



# 安全漏洞實例

:ref. 《Effective Java》 item56

```
public final class Period implements Serializable {
    private final Date start;
    private final Date end;

    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end    = new Date(end.getTime());
        if (this.start.compareTo(this.end) > 0)
            throw new IllegalArgumentException(start + " > " + end);
    }

    public Date start () { return (Date) start.clone(); }
    public Date end   () { return (Date) end.clone();   }
    public String toString() { return start + " - " + end; }

    public static void main(String[] args)
        throws ClassNotFoundException, IOException {
        Period p = new Period(new Date(61,8,28,1,2,3), new Date(67,3,28,4,5,6));
        System.out.println(p);
        ObjectOutputStream out =
            new ObjectOutputStream(new FileOutputStream("Period.out"));
        out.writeObject(p);
        out.close();
    }
}
```

> println結果：Thu Sep 28 01:02:03 CDT 1961 - Fri Apr 28 04:05:06 CST 1967



# 安全漏洞

:ref. 《Effective Java》 item56

➤ tdump Period.out

```

000000: AC ED 00 05 73 72 00 06 50 65 72 69 6F 64 73 F8 秒..sr..Periods懷
000010: F7 A8 FD BA 46 E4 02 00 02 4C 00 03 65 6E 64 74 味慚....L..endt
000020: 00 10 4C 6A 61 76 61 2F 75 74 69 6C 2F 44 61 74 ..Ljava/util/Dat
000030: 65 3B 4C 00 05 73 74 61 72 74 71 00 7E 00 01 78 e;L..startq.~..x
000040: 70 73 72 00 0E 6A 61 76 61 2E 75 74 69 6C 2E 44 psr..java.util.D
000050: 61 74 65 68 6A 81 01 4B 59 74 19 03 00 00 78 70 atehj..KYt....xp
000060: 77 08 FF FF FF EC 4D 77 1D 50 78 73 71 00 7E 00 w....閨w.Pxsq.~.
000070: 03 77 08 FF FF FF C3 4D 38 30 78 78 00 00 00 00 .w....騎80xx....

```

➤ 刻意製造一個模擬 Period.out 內容的 byte stream:

```

private static final byte[] serializedForm = new byte[] {
    (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x06,
    0x50, 0x65, 0x72, 0x69, 0x6f, 0x64, 0x40, 0x7e, (byte)0xf8,
    0x2b, 0x4f, 0x46, (byte)0xc0, (byte)0xf4, 0x02, 0x00, 0x02,
    0x4c, 0x00, 0x03, 0x65, 0x6e, 0x64, 0x74, 0x00, 0x10, 0x4c,
    0x6a, 0x61, 0x76, 0x61, 0x2f, 0x75, 0x74, 0x69, 0x6c, 0x2f,
    0x44, 0x61, 0x74, 0x65, 0x3b, 0x4c, 0x00, 0x05, 0x73, 0x74,
    0x61, 0x72, 0x74, 0x71, 0x00, 0x7e, 0x00, 0x01, 0x78, 0x70,
    0x73, 0x72, 0x00, 0x0e, 0x6a, 0x61, 0x76, 0x61, 0x2e, 0x75,
    0x74, 0x69, 0x6c, 0x2e, 0x44, 0x61, 0x74, 0x65, 0x68, 0x6a,
    (byte)0x81, 0x01, 0x4b, 0x59, 0x74, 0x19, 0x03, 0x00, 0x00,
    0x78, 0x70, 0x77, 0x08, 0x00, 0x00, 0x00, 0x66, (byte)0xdf,
    0x6e, 0x1e, 0x00, 0x78, 0x73, 0x71, 0x00, 0x7e, 0x00, 0x03,
    0x77, 0x08, 0x00, 0x00, 0x00, (byte)0xd5, 0x17, 0x69, 0x22,
    0x00, 0x78 };

```



# 安全漏洞

:ref. 《Effective Java》 item56

➤ tdump Period.out

AC	ED	00	05	73	72	00	06	50	65	72	69	6F	64	73	F8
F7	A8	FD	BA	46	E4	02	00	02	4C	00	03	65	6E	64	74
00	10	4C	6A	61	76	61	2F	75	74	69	6C	2F	44	61	74
65	3B	4C	00	05	73	74	61	72	74	71	00	7E	00	01	78
70	73	72	00	0E	6A	61	76	61	2E	75	74	69	6C	2E	44
61	74	65	68	6A	81	01	4B	59	74	19	03	00	00	78	70
77	08	FF	FF	FF	EC	4D	77	1D	50	78	73	71	00	7E	00
03	77	08	FF	FF	FF	C3	4D	38	30	78	78				

➤ 刻意製造一個模擬 Period.out 內容的 byte stream

ac	ed	00	05	73	72	00	06	50	65	72	69	6f	64	40	7e
f8	2b	4f	46	c0	f4	02	00	02	4c	00	03	65	6e	64	74
00	10	4c	6a	61	76	61	2f	75	74	69	6c	2f	44	61	74
65	3b	4c	00	05	73	74	61	72	74	71	00	7e	00	01	78
70	73	72	00	0e	6a	61	76	61	2e	75	74	69	6c	2e	44
61	74	65	68	6a	81	01	4b	59	74	19	03	00	00	78	70
77	08	00	00	00	66	df	6e	1e	00	78	73	71	00	7e	00
03	77	08	00	00	00	d5	17	69	22	00	78				

➤ println結果：Fri Jan 01 12:00:00 PST 1999 - Sun Jan 01 12:00:00 PST 1984. (違反約束條件)



# 防堵安全漏洞

:ref. 《Effective Java》 item56

```
private void readObject(ObjectInputStream s)
    throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    // Check that our invariants are satisfied
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start + " after " + end);
}
```

➤ 另一個安全漏洞見《Effective Java》ch10, p227



# Serialization in MFC

: static type system 的先天障礙

C++ 是一個 **static type system** 語言，程式所用的所有 classes 都必須在編譯期可見。因此 C++ library 實踐 Object Serialization 的最大困難在於，deSerialization 過程中如何根據 byte stream 復創物件（注意：C++ 創建物件一定要透過 new operator）。

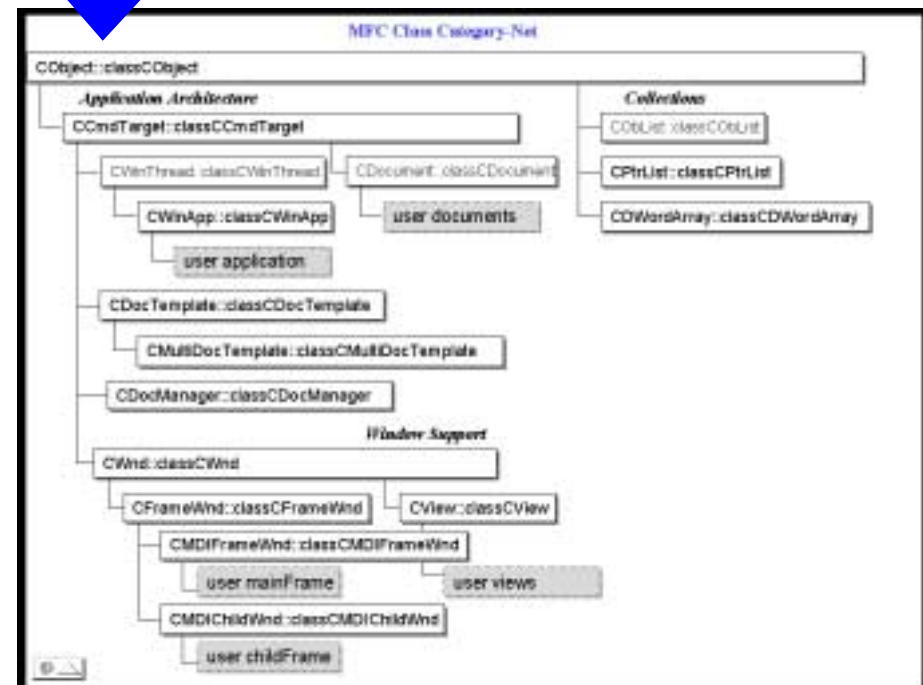
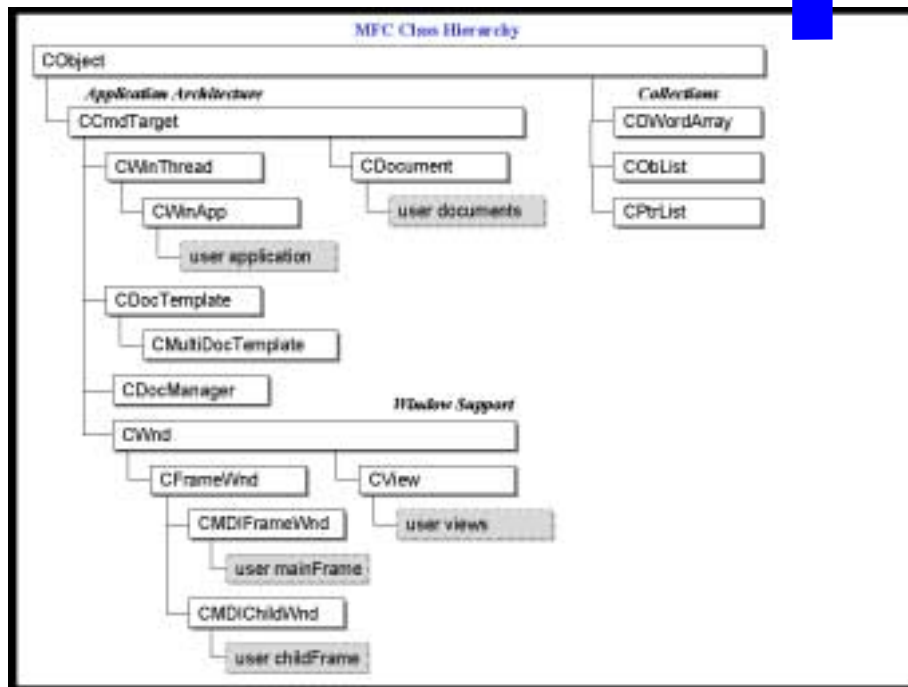




# Serialization in MFC

## : 埋藏暗樁

- 運用 static members, global object's constructor, macros, MFC 針對每一個 MFC classes 和每一個想要 serialization 功能的 MFC app. classes, 在 MFC App. 中埋藏了一些暗樁, 以及整片 "暗樁森林"。







# Serialization in MFC

## : 暗樁形式

**CRuntimeClass**  
**CFoo::classCFoo;**

"CFoo"
m_nObjectSize
m_wSchema
m_pfnCreateObject
m_pBaseClass
m_pNextClass

static CObject* CFoo::CreateObject()  
{  
    return new CFoo;  
}



# Serialization in MFC

: 無 default serialization form, 需有明確的 **Serialize()**

## ➤ 暗樁的形成

```
class CFoo : public CObject
{
    DECLARE_SERIAL(CFoo);
};

IMPLEMENT_SERIAL(CFoo, CObject, 1);
```

## ➤ 負責 serialization/deSerialization

```
void CFoo::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);
    if (ar.IsStoring()) {
        ...
    }
    else {
        ...
    }
}
```

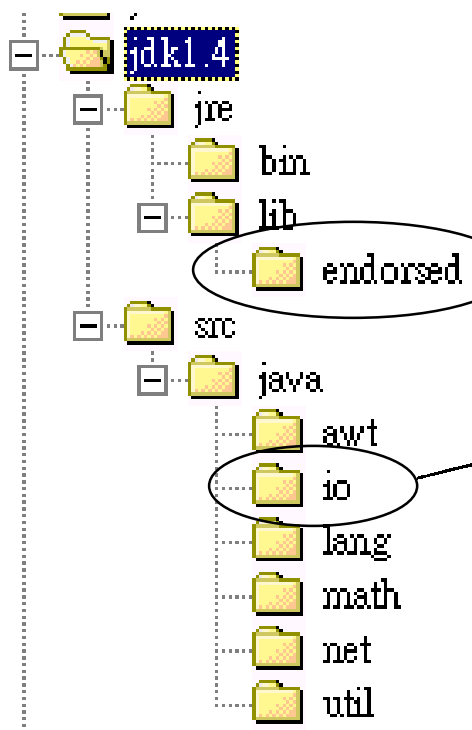


# Java Library源碼修改經驗

- c: \jdk1.4\src\java\io\ObjectStreamClass.java  
這是待改源碼（備份為宜）。修改後編譯，產生*.class。
- c: \jdk1.4\jre\lib\rt.jar  
這是Java Library classes（過程中不受影響）
- c: \myprog\Test.java  
這是測試程式
- c: \jdk1.4\jre\lib\endorsed（背書、簽署、贊同、認可）  
新增這個子目錄，classloader將優先從這兒讀取.jar。
- 把ObjectStreamClass*.class 搬移到 \temp，壓縮為xxx.jar（夾帶路徑 java\io）並將它複製到上述的endorsed
- 如此一來測試程式便可正確用到修改後的class



# Java Library源碼修改經驗



把修改後的.java 編譯為.class，  
壓縮為 xxx.jar（帶路徑 java\io），  
複製到endorsed。  
即可被class loader優先讀取。



## 更多資訊

- 《*Java Object Serialization Specification*》  
by Sun Microsystems Inc.
- Java source code :  
`io.FileOutputStream, io.ObjectOutputStream, io.ObjectStreamClass,`  
`util.LinkedList, util.ArrayList, lang.Integer, lang.Number,`  
`lang.reflect.Constructor, lang.reflect.Method...`
- 《*Effective Java*》 chap10 "Serialization", by Joshua Bloch.
- 《*Thinking in Java*》 chap11: Java I/O System; chap12:RTTI.
- 《深入淺出 *MFC*》 第8章 "Doc/View 深入探討", by 侯捷
- MFC source code

本次研討詳細內容將整理為文，2003第三季刊於台北《Run!PC》和北京《程序員》，並於次月以 PDF 開放於侯捷網站（<http://www.jjhhou.com>）。

# Thank You!



Java[™]

Sun Microsystems