2015. java学习交流群

8918 1289

每天有免费的Java学习课堂

——学习Java就是这么简单

——为Java而燃烧——

Java 语言编码规范(Java Code Conventions)

译者　晨光（Morning）
搜集整理 ：华竹技术实验室 http://sinoprise.com
简介 ：
本文档讲述了 Java 语言的编码规范，较之陈世忠先生《c++编码规范》的浩繁详尽，此文当属短小精悍了。而其中所列之各项条款，从编码风格，到注意事项，不单只 Java，对于其他语言，也都很有借鉴意义。因为简短，所以易记，大家不妨将此作为 handbook，常备案头，逐一对验。

目录

1 介绍(Introduction)

1.1 为什么要有编码规范(Why Have Code Conventions)

编码规范对于程序员而言尤为重要，有以下几个原因：

- 一个软件的生命周期中，80%的花费在于维护
- 几乎没有任何一个软件，在其整个生命周期中，均由最初的开发人员来维护
- 编码规范可以改善软件的可读性，可以让程序员尽快而彻底地理解新的代码
- 如果你将源码作为产品发布，就需要确任它是否被很好的打包并且清晰无误，一如你已构建的其它任何产品

为了执行规范，每个软件开发人员必须一致遵守编码规范。每个人。

1.2 版权声明(Acknowledgments)

本文档反映的是 Sun MicroSystems 公司，Java 语言规范中的编码标准部分。主要贡献者包括：Peter King，Patrick Naughton，Mike DeMoney，Jonni Kanerva，Kathy Walrath 以及 Scott Hommel。

本文档现由 Scott Hommel 维护，有关评论意见请发至 shommel@eng.sun.com

2 文件名(File Names)

这部分列出了常用的文件名及其后缀。

2.1 文件后缀(File Suffixes)

Java 程序使用下列文件后缀：

文件类别	文件后缀
Java 源文件	.java
Java 字节码文件	.class

2.2 常用文件名(Common File Names)

常用的文件名包括：

文件名	用途
GNUmakefile	makefiles 的首选文件名。我们采用 gnumake 来创建（build）软件。
README	概述特定目录下所含内容的文件的首选文件名

3 文件组织(File Organization)

一个文件由被空行分割而成的段落以及标识每个段落的可选注释共同组成。超过 2000 行的程序难以阅读，应该尽量避免。"Java 源文件范例"提供了一个布局合理的 Java 程序范例。

3.1 Java 源文件(Java Source Files)

每个 Java 源文件都包含一个单一的公共类或接口。若私有类和接口与一个公共类相关联，可以将它们和公共类放入同一个源文件。公共类必须是这个文件中的第一个类或接口。

Java 源文件还遵循以下规则：

- 开头注释（参见"开头注释"）
- 包和引入语句（参见"包和引入语句"）
- 类和接口声明（参见"类和接口声明"）

3.1.1 开头注释(Beginning Comments)

所有的源文件都应该在开头有一个 C 语言风格的注释，其中列出类名、版本信息、日期和版权声明：

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

3.1.2 包和引入语句(Package and Import Statements)

在多数 Java 源文件中，第一个非注释行是包语句。在它之后可以跟引入语句。例如：

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

3.1.3 类和接口声明(Class and Interface Declarations)

下表描述了类和接口声明的各个部分以及它们出现的先后次序。参见"Java源文件范例"中一个包含注释的例子。

	类/接口声明的各部分	注解
1	类/接口文档注释 (/**.....*/)	该注释中所需包含的信息，参见"文档注释"
2	类或接口的声明	
3	类/接口实现的注释	该注释应包含任何有关整个类或接口的信息，而这些信息又不适合

	(/*……*/)如果有必要的话	作为类/接口文档注释。
4	类的(静态)变量	首先是类的公共变量，随后是保护变量，再后是包一级别的变量(没有访问修饰符，access modifier)，最后是私有变量。
5	实例变量	首先是公共级别的，随后是保护级别的，再后是包一级别的(没有访问修饰符)，最后是私有级别的。
6	构造器	
7	方法	这些方法应该按功能，而非作用域或访问权限，分组。例如，一个私有的类方法可以置于两个公有的实例方法之间。其目的是为了更便于阅读和理解代码。

4 缩进排版(Indentation)

4 个空格常被作为缩进排版的一个单位。缩进的确切解释并未详细指定(空格 vs. 制表符)。一个制表符等于 8 个空格(而非 4 个)。

4.1 行长度(Line Length)

尽量避免一行的长度超过 80 个字符，因为很多终端和工具不能很好处理之。

注意：用于文档中的例子应该使用更短的行长，长度一般不超过 70 个字符。

4.2 换行(Wrapping Lines)

当一个表达式无法容纳在一行内时，可以依据如下一般规则断开之：

- 在一个逗号后面断开
- 在一个操作符前面断开
- 宁可选择较高级别(higher-level)的断开，而非较低级别(lower-level)的断开
- 新的一行应该与上一行同一级别表达式的开头处对齐
- 如果以上规则导致你的代码混乱或者使你的代码都堆挤在右边，那就代之以缩进 8 个空格。

以下是断开方法调用的一些例子：

```
someMethod(longExpression1, longExpression2, longExpression3,
                longExpression4, longExpression5);

var = someMethod1(longExpression1,
                          someMethod2(longExpression2,
                                      longExpression3));
```

以下是两个断开算术表达式的例子。前者更好，因为断开处位于括号表达式的外边，这是个较高级别的断开。

```
  longName1 = longName2 * (longName3 + longName4 - longName5)
                    + 4 * longname6; //PREFFER

  longName1 = longName2 * (longName3 + longName4
                                        - longName5) + 4 * longname6;
//AVOID
```

以下是两个缩进方法声明的例子。前者是常规情形。后者若使用常规的缩进方式将会使第二行和第三行移得很靠右，所以代之以缩进 8 个空格

```
  //CONVENTIONAL INDENTATION
  someMethod(int anArg, Object anotherArg, String yetAnotherArg,
                    Object andStillAnother) {
    ...
  }

  //INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
  private static synchronized horkingLongMethodName(int anArg,
          Object anotherArg, String yetAnotherArg,
          Object andStillAnother) {
    ...
  }
```

if 语句的换行通常使用 8 个空格的规则，因为常规缩进(4 个空格)会使语句体看起来比较费劲。比如：

```
  //DON’T USE THIS INDENTATION
  if ((condition1 && condition2)
      || (condition3 && condition4)
      ||!(condition5 && condition6)) { //BAD WRAPS
      doSomethingAboutIt();             //MAKE THIS LINE EASY TO MISS
  }

  //USE THIS INDENTATION INSTEAD
  if ((condition1 && condition2)
          || (condition3 && condition4)
          ||!(condition5 && condition6)) {
      doSomethingAboutIt();
  }

  //OR USE THIS
  if ((condition1 && condition2) || (condition3 && condition4)
          ||!(condition5 && condition6)) {
```

```
    doSomethingAboutIt();
}
```

这里有三种可行的方法用于处理三元运算表达式：

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

5 注释(Comments)

Java 程序有两类注释：实现注释(implementation comments)和文档注释(document comments)。实现注释是那些在 C++中见过的，使用/*...*/和//界定的注释。文档注释(被称为"doc comments")是 Java 独有的，并由/**...*/界定。文档注释可以通过 javadoc 工具转换成 HTML 文件。

实现注释用以注释代码或者实现细节。文档注释从实现自由(implementation-free)的角度描述代码的规范。它可以被那些手头没有源码的开发人员读懂。

注释应被用来给出代码的总括，并提供代码自身没有提供的附加信息。注释应该仅包含与阅读和理解程序有关的信息。例如，相应的包如何被建立或位于哪个目录下之类的信息不应包括在注释中。

在注释里，对设计决策中重要的或者不是显而易见的地方进行说明是可以的，但应避免提供代码中已清晰表达出来的重复信息。多余的的注释很容易过时。通常应避免那些代码更新就可能过时的注释。

注意：频繁的注释有时反映出代码的低质量。当你觉得被迫要加注释的时候，考虑一下重写代码使其更清晰。

注释不应写在用星号或其他字符画出来的大框里。注释不应包括诸如制表符和回退符之类的特殊字符。

5.1 实现注释的格式(Implementation Comment Formats)

程序可以有 4 种实现注释的风格：块(block)、单行(single-line)、尾端(trailing)和行末(end-of-line)。

5.1.1 块注释(Block Comments)

块注释通常用于提供对文件，方法，数据结构和算法的描述。块注释被置于每个文件的开始处以及每个方法之前。它们也可以被用于其他地方，比如方法内部。在功能和方法内部的块注释应该和它们所描述的代码具有一样的缩进格式。

块注释之首应该有一个空行，用于把块注释和代码分割开来，比如：

```
/*
 * Here is a block comment.
 */
```

块注释可以以/*-开头，这样 indent(1)就可以将之识别为一个代码块的开始，而不会重排它。

```
/*-
  * Here is a block comment with some very special
  * formatting that I want indent(1) to ignore.
  *
  *    one
  *        two
  *            three
  */
```

注意：如果你不使用 indent(1)，就不必在代码中使用/*-，或为他人可能对你的代码运行 indent(1)作让步。

参见"文档注释"

5.1.2 单行注释(Single-Line Comments)

短注释可以显示在一行内，并与其后的代码具有一样的缩进层级。如果一个注释不能在一行内写完，就该采用块注释(参见"块注释")。单行注释之前应该有一个空行。以下是一个Java代码中单行注释的例子：

```
if (condition) {

  /* Handle the condition. */
  ...
}
```

5.1.3 尾端注释(Trailing Comments)

极短的注释可以与它们所要描述的代码位于同一行，但是应该有足够的空白来分开代码和注释。若有多个短注释出现于大段代码中，它们应该具有相同的缩进。

以下是一个 Java 代码中尾端注释的例子：

```
if (a == 2) {
    return TRUE;              /* special case */
} else {
    return isPrime(a);         /* works only for odd a */
}
```

5.1.4 行末注释(End-Of-Line Comments)

注释界定符"//"，可以注释掉整行或者一行中的一部分。它一般不用于连续多行的注释文本；然而，它可以用来注释掉连续多行的代码段。以下是所有三种风格的例子：

```
if (foo > 1) {

    // Do a double-flip.
    ...
}
else {
    return false;          // Explain why here.
}

//if (bar > 1) {
//
//    // Do a triple-flip.
//    ...
//}
//else {
//    return false;
//}
```

5.2 文档注释(Documentation Comments)

注意：此处描述的注释格式之范例，参见"Java源文件范例"

若想了解更多，参见"How to Write Doc Comments for Javadoc"，其中包含了有关文档注释标记的信息(@return, @param, @see)：

http://java.sun.com/javadoc/writingdoccomments/index.html

若想了解更多有关文档注释和 javadoc 的详细资料，参见 javadoc 的主页：

http://java.sun.com/javadoc/index.html

文档注释描述 Java 的类、接口、构造器，方法，以及字段(field)。每个文档注释都会被置于注释定界符/**...*/之中，一个注释对应一个类、接口或成员。该注释应位于声明之前：

```
/**
  * The Example class provides ...
  */
public class Example { ...
```

注意顶层(top-level)的类和接口是不缩进的，而其成员是缩进的。描述类和接口的文档注释的第一行(/**)不需缩进；随后的文档注释每行都缩进 1 格(使星号纵向对齐)。成员，包括构造函数在内，其文档注释的第一行缩进 4 格，随后每行都缩进 5 格。

若你想给出有关类、接口、变量或方法的信息，而这些信息又不适合写在文档中，则可使用实现块注释(见 5.1.1)或紧跟在声明后面的单行注释(见 5.1.2)。例如，有关一个类实现的细节，应放入紧跟在类声明后面的实现块注释中，而不是放在文档注释中。

文档注释不能放在一个方法或构造器的定义块中，因为 Java 会将位于文档注释之后的第一个声明与其相关联。

6 声明(Declarations)

6.1 每行声明变量的数量(Number Per Line)

推荐一行一个声明，因为这样以利于写注释。亦即，

```
int level;  // indentation level
int size;   // size of table
```

要优于，

```
int level, size;
```

不要将不同类型变量的声明放在同一行，例如：

```
int foo,  fooarray[];   //WRONG!
```

注意：上面的例子中，在类型和标识符之间放了一个空格，另一种被允许的替代方式是使用制表符：

```
int         level;         // indentation level
int         size;          // size of table
Object      currentEntry;  // currently selected table entry
```

6.2 初始化(Initialization)

尽量在声明局部变量的同时初始化。唯一不这么做的理由是变量的初始值依赖于某些先前发生的计算。

6.3 布局(Placement)

只在代码块的开始处声明变量。（一个块是指任何被包含在大括号"{"和"}"中间的代码。）不要在首次用到该变量时才声明之。这会把注意力不集中的程序员搞糊涂，同时会妨碍代码在该作用域内的可移植性。

```
void myMethod() {
    int int1 = 0;         // beginning of method block

    if (condition) {
        int int2 = 0;     // beginning of "if" block
        ...
    }
}
```

该规则的一个例外是 for 循环的索引变量

```
for (int i = 0; i < maxLoops; i++) { ... }
```

避免声明的局部变量覆盖上一级声明的变量。例如，不要在内部代码块中声明相同的变量名：

```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;     // AVOID!
        ...
    }
    ...
}
```

6.4 类和接口的声明(Class and Interface Declarations)

当编写类和接口是，应该遵守以下格式规则：

- 在方法名与其参数列表之前的左括号"("间不要有空格
- 左大括号"{"位于声明语句同行的末尾
- 右大括号"}"另起一行，与相应的声明语句对齐，除非是一个空语句，"}"应紧跟在"{"之后

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod() {}

    ...
}
```

- 方法与方法之间以空行分隔

7 语句(Statements)

7.1 简单语句(Simple Statements)

每行至多包含一条语句，例如：

```
argv++;       // Correct
argc--;       // Correct
argv++; argc--;       // AVOID!
```

7.2 复合语句(Compound Statements)

复合语句是包含在大括号中的语句序列，形如"{ 语句 }"。例如下面各段。

- 被括其中的语句应该较之复合语句缩进一个层次
- 左大括号"{"应位于复合语句起始行的行尾；右大括号"}"应另起一行并与复合语句首行对齐。
- 大括号可以被用于所有语句，包括单个语句，只要这些语句是诸如 if-else 或 for 控制结构的一部分。这样便于添加语句而无需担心由于忘了加括号而引入 bug。

7.3 返回语句(return Statements)

一个带返回值的 return 语句不使用小括号"()"，除非它们以某种方式使返回值更为显见。例如：

```
return;
```

```
return myDisk.size();

return (size ? size : defaultSize);
```

7.4 if，if-else，if else-if else 语句(if, if-else, if else-if else Statements)

if-else 语句应该具有如下格式：

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
    statements;
} else{
    statements;
}
```

注意：if 语句总是用"{"和"}"括起来，避免使用如下容易引起错误的格式：

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
    statement;
```

7.5 for 语句(for Statements)

一个 for 语句应该具有如下格式：

```
for (initialization; condition; update) {
    statements;
}
```

一个空的 for 语句(所有工作都在初始化，条件判断，更新子句中完成）应该具有如下格式：

```
for (initialization; condition; update);
```

当在 for 语句的初始化或更新子句中使用逗号时，避免因使用三个以上变量，而导致复杂度提高。若需要，可以在 for 循环之前(为初始化子句)或 for 循环末尾(为更新子句)使用单独的语句。

7.6 while 语句(while Statements)

一个 while 语句应该具有如下格式

```
while (condition) {
    statements;
}
```

一个空的 while 语句应该具有如下格式：

```
while (condition);
```

7.7 do-while 语句(do-while Statements)

一个 do-while 语句应该具有如下格式：

```
do {
    statements;
} while (condition);
```

7.8 switch 语句(switch Statements)

一个 switch 语句应该具有如下格式：

```
switch (condition) {
case ABC:
    statements;
    /* falls through */
case DEF:
    statements;
    break;

case XYZ:
    statements;
    break;
```

```
default:
    statements;
    break;
}
```

每当一个 case 顺着往下执行时(因为没有 break 语句)，通常应在 break 语句的位置添加注释。上面的示例代码中就包含注释/* falls through */。

7.9 try-catch 语句(try-catch Statements)

一个 try-catch 语句应该具有如下格式：

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}
```

一个 try-catch 语句后面也可能跟着一个 finally 语句，不论 try 代码块是否顺利执行完，它都会被执行。

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
```

8 空白(White Space)

8.1 空行(Blank Lines)

空行将逻辑相关的代码段分隔开，以提高可读性。

下列情况应该总是使用两个空行：

- 一个源文件的两个片段(section)之间
- 类声明和接口声明之间

下列情况应该总是使用一个空行：

- 两个方法之间
- 方法内的局部变量和方法的第一条语句之间
- 块注释（参见"5.1.1"）或单行注释（参见"5.1.2"）之前
- 一个方法内的两个逻辑段之间，用以提高可读性

8.2 空格(Blank Spaces)

下列情况应该使用空格：

- 一个紧跟着括号的关键字应该被空格分开，例如：

```
while (true) {
    ...
}
```

注意：空格不应该置于方法名与其左括号之间。这将有助于区分关键字和方法调用。
- 空白应该位于参数列表中逗号的后面
- 所有的二元运算符，除了"."，应该使用空格将之与操作数分开。一元操作符和操作数之间不因该加空格，比如：负号("-")、自增("++")和自减("--")。例如：

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
    n++;
}
printSize("size is " + foo + "\n");
```

- for 语句中的表达式应该被空格分开，例如：

```
for (expr1; expr2; expr3)
```

- 强制转型后应该跟一个空格，例如：

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

9 命名规范(Naming Conventions)

命名规范使程序更易读，从而更易于理解。它们也可以提供一些有关标识符功能的信息，以助于理解代码，例如，不论它是一个常量，包，还是类。

标识符类型	命名规则	例子
包(Packages)	一个唯一包名的前缀总是全部小写的 ASCII 字母并且是一个顶级域名，通常是 com，edu，gov，mil，net，org，或 1981 年 ISO 3166	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese

	标准所指定的标识国家的英文双字符代码。包名的后续部分根据不同机构各自内部的命名规范而不尽相同。这类命名规范可能以特定目录名的组成来区分部门(department)，项目(project)，机器(machine)，或注册名(login names)。	
类(Classes)	命名规则：类名是个一名词，采用大小写混合的方式，每个单词的首字母大写。尽量使你的类名简洁而富于描述。使用完整单词，避免缩写词(除非该缩写词被更广泛使用，像 URL，HTML)	class Raster; class ImageSprite;
接口(Interfaces)	命名规则：大小写规则与类名相似	interface RasterDelegate; interface Storing;

方法(Methods)	方法名是一个动词，采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写。	run();
		runFast();
		getBackground();

变量(Variables)	除了变量名外，所有实例，包括类，类常量，均采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写。变量名不应以下划线或美元符号开头，尽管这在语法上是允许的。变量名应简短且富于描述。变量名的选用应该易于记忆，即，能够指出其用途。尽量避免单个字符的变量名，除非是一次性的临时变量。临时变量通常被取名为 i，j，k，m 和 n，它们一般用于整型；c，d，e，它们一般用于字符型。	char c; int i; float myWidth;
实例变量(Instance Variables)	大小写规则和变量名相似，除了前面需要一个下划线	int _employeeId; String _name; Customer _customer;

常量(Constants)	类常量和 ANSI 常量的声明，应该全部大写，单词间用下划线隔开。(尽量避免 ANSI 常量，容易引起错误)	static final int MIN_WIDTH = 4;
		static final int MAX_WIDTH = 999;
		static final int GET_THE_CPU = 1;

10 编程惯例(Programming Practices)

10.1 提供对实例以及类变量的访问控制(Providing Access to Instance and Class Variables)

若没有足够理由，不要把实例或类变量声明为公有。通常，实例变量无需显式的设置(set)和获取(gotten)，通常这作为方法调用的边缘效应 (side effect)而产生。

一个具有公有实例变量的恰当例子，是类仅作为数据结构，没有行为。亦即，若你要使用一个结构(struct)而非一个类(如果 java 支持结构的话)，那么把类的实例变量声明为公有是合适的。

10.2 引用类变量和类方法(Referring to Class Variables and Methods)

避免用一个对象访问一个类的静态变量和方法。应该用类名替代。例如：

```
classMethod();              //OK
AClass.classMethod();       //OK
anObject.classMethod();     //AVOID!
```

10.3 常量(Constants)

位于 for 循环中作为计数器值的数字常量，除了-1,0 和 1 之外，不应被直接写入代码。

10.4 变量赋值(Variable Assignments)

避免在一个语句中给多个变量赋相同的值。它很难读懂。例如：

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

不要将赋值运算符用在容易与相等关系运算符混淆的地方。例如：

```
if (c++ = d++) {        // AVOID! (Java disallows)
    ...
}
```

应该写成

```
if ((c++ = d++) != 0) {
  ...
}
```

不要使用内嵌(embedded)赋值运算符试图提高运行时的效率，这是编译器的工作。例如：

```
d = (a = b + c) + r;        // AVOID!
```

应该写成

```
a = b + c;
d = a + r;
```

10.5 其它惯例(Miscellaneous Practices)

10.5.1 圆括号(Parentheses)

一般而言，在含有多种运算符的表达式中使用圆括号来避免运算符优先级问题，是个好方法。即使运算符的优先级对你而言可能很清楚，但对其他人未必如此。你不能假设别的程序员和你一样清楚运算符的优先级。

```
if (a == b && c == d)     // AVOID!
if ((a == b) && (c == d))  // RIGHT
```

10.5.2 返回值(Returning Values)

设法让你的程序结构符合目的。例如：

```
if (booleanExpression) {
    return true;
} else {
    return false;
}
```

应该代之以如下方法：

```
return booleanExpression;
```

类似地：

```
if (condition) {
    return x;
}
return y;
```

应该写做：

```
return (condition ? x : y);
```

10.5.3 条件运算符"?"前的表达式(Expressions before '?' in the Conditional Operator)

如果一个包含二元运算符的表达式出现在三元运算符" ? : "的"?"之前，那么应该给表达式添上一对圆括号。例如：

```
(x >= 0) ? x : -x;
```

10.5.4 特殊注释(Special Comments)

在注释中使用 XXX 来标识某些未实现(bogus)的但可以工作(works)的内容。用 FIXME 来标识某些假的和错误的内容。

11 代码范例(Code Examples)

11.1 Java 源文件范例(Java Source File Example)

下面的例子，展示了如何合理布局一个包含单一公共类的Java源程序。接口的布局与其相似。更多信息参见"类和接口声明"以及"文挡注释"。

```
/*
 * @(#)Blah.java         1.82 99/03/18
 *
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information").  You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 */


package java.blah;

import java.blah.blahdy.BlahBlah;

/**
 * Class description goes here.
 *
 * @version     1.82 18 Mar 1999
 * @author      Firstname Lastname
 */
public class Blah extends SomeClass {
    /* A class implementation comment can go here. */
```

```
    /** classVar1 documentation comment */
    public static int classVar1;

    /**
     * classVar2 documentation comment that happens to be
     * more than one line long
     */
    private static Object classVar2;

    /** instanceVar1 documentation comment */
    public Object instanceVar1;

    /** instanceVar2 documentation comment */
    protected int instanceVar2;

    /** instanceVar3 documentation comment */
    private Object[] instanceVar3;

    /**
     * ...constructor Blah documentation comment...
     */
    public Blah() {
        // ...implementation goes here...
    }

    /**
     * ...method doSomething documentation comment...
     */
    public void doSomething() {
        // ...implementation goes here...
    }

    /**
     * ...method doSomethingElse documentation comment...
     * @param someParam description
     */
    public void doSomethingElse(Object someParam) {
        // ...implementation goes here...
    }
}
```

侯捷觀點

JDK 1.5 的泛型實現

— Generics in JDK 1.5 —

北京《程序員》2004/09
台北《Run!PC》2004/09

作者簡介：侯捷，資訊教育、專欄執筆、大學教師。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，使用過 Java Collections。
- 本文適用工具：JDK1.5
- 本文程式源碼可至侯捷網站下載
  http://www.jjhou.com/javatwo-2004-reflection-and-generics-in-jdk15-sample.ZIP
- 本文是 JavaTwo-2004 技術研討會同名講題之部分內容書面整理。
- 關鍵術語：
  persistence（永續性、持久性）
  serialization（序列化、次第讀寫）
  generics（泛型）
  polymorphism（多型）

全文提要

泛型技術與 Sun JDK 的淵源可追溯自 JDK1.3。但無論 JDK 1.3 或 JDK1.4，都只是以編譯器外掛附件的方式來支援泛型語法，並且 Java 標準程式庫未曾針對泛型全

面改寫。而今 JDK1.5 正式納入泛型。本文討論 JDK1.5 的泛型實現，包括如何使用及自訂 generic classes and algorithms，其中若干語法異於 JDK 1.3 和 1.4。

我曾經在JavaTwo 2002大會上針對泛型技術給出一個講題，並將內容整理爲《Java 泛型技術之發展》一文（http://www.jjhou.com/javatwo-2002.htm）。該文所談的 Java 泛型語法以及 Java 泛型技術之內部實作技術，在今天（被 JDK 1.5 正式納入）依然適用。但由於有了若干小變化，並且由於 Java 標準程式庫的全面改寫，使我認爲有必要再整理這篇文章，讓讀者輕鬆地在 JDK 1.5 中繼續悠遊「泛型」技術。

閱讀本文之前，如果自覺基礎不夠，可以補充閱讀適才提到的《Java 泛型技術之發展》，那是一篇非常完整的文章，可助您完整認識泛型技術的來龍去脈。

Sun JDK 的泛型發展歷史要從 1.3 版說起。該版本配合 GJ，正式進入泛型殿堂。所謂 GJ 是 "Generic Java" 的縮寫，是一個支援泛型的 Java 編譯器補充件，可謂 Java 泛型技術的先趨。隨後，泛型議題正式成爲 JSR #14，其技術基礎便是源自 GJ。JDK1.4 搭配 JSR14 提供的外掛附件，使泛型技術在 Java 世界從妾身未明的身份扶正而爲眾所屬目的焦點。今天，JDK1.5 終於內建泛型特性，不僅編譯器不再需要任何外力（外掛附件）的幫助，整個 Java 標準程式庫也被翻新（retrofit），許多角落針對泛型做了改寫。

讓我們把帶有「參數化型別」（parameterized types）的 classes 稱爲 generic classes，把帶有「參數化型別」的 methods 稱爲 generic algorithms，那麼，對眾多 Java 程式員而言，泛型帶來的影響不外乎以下四點，稍後逐一說明。

- 如何使用 generic classes
- 如何使用 generic algorithms
- 如何自訂 generic classes
- 如何自訂 generic algorithms

在此先提醒您，運用泛型時，加上 `-Xlint:unchecked` 編譯選項，可讓編譯器幫助我們檢查潛在的型別轉換問題。

何謂 Generic Classes

Generic classes 的最大宗運用是 collections（群集），也就是實作各種資料結構（例如 list, map, set, hashtable）的那些 classes。也有人稱它們爲容器（containers）。這些容器被設計用來存放 `Object`-derived 元素。而由於 Java 擁有單根繼承體系，任何 Java classes 都繼承自 `java.lang.Object`，因此任何 Java objects 都可以被放進上述各種容器。換句話說 Java 容器是一種異質容器，從「泛型」的字面意義來說，其實這（原本的設計）才是「泛型」。

然而有時候，而且是大半時候，我們不希望容器元素如此異質化。我們多半希望使用同質容器。即使用於多型（polymorphism），我們也希望至少相當程度地規範容器，令其元素型別爲「帶有某種約束」的 base class。例如面對一個準備用來放置各種形狀（圓圈、橢圓、矩形、四方形、三角形…）的容器，如果我們能夠告知這個容器其每個元素都必須是 `Shape`-derived objects，將相當有助於程式的可讀性，並減少錯誤，容易除錯，甚至可避免一大堆轉型（cast）動作。

Java 同質容器的語法如下，其中角括號（<>）的用法和 C++完全相同，角括號之內的指定型別，就是同質容器的元素型別，如圖 1。

```
ArrayList<String> strList = new ArrayList<String>();
strList.add("zero");
strList.add("one");
strList.add("two");
strList.add("five");
System.out.println(strList);  // [zero, one, two, five]
```

圖 1 / 同質容器的用法。角括號（<>）內就是元素型別。

下面是另一個實例，程式員要求容器內的每一個元素都必須是「一種形狀」，這是一種「多型」應用，如圖 2。這些泛型語法自 JDK 1.3+GJ 以來不曾改變過。

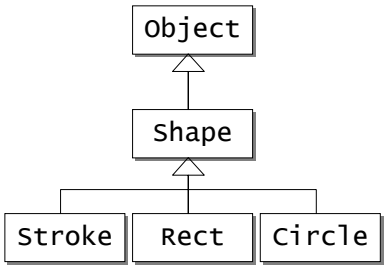


圖 2a / 典型的 "Shape" 多型繼承體系。

```
//假設 Stroke, Rect, Circle 皆繼承自 Shape
LinkedList<Shape> sList = new LinkedList<Shape>();
sList.add(new Stroke(...));
sList.add(new Rect(...));
sList.add(new Circle(...));
```

圖 2b / 令容器內含各種 Shape 元素，並加入一個 Stroke，一個 Rect 和一個 Circle。

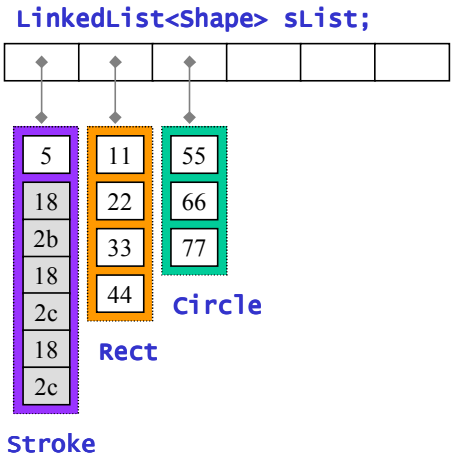


圖 2c / 圖 2b 程式碼所製造的結果。

Boxing 和 Un-boxing 帶來的影響

前面曾經說過，任何 Java objects 都可以被放進各種容器內。但是 Java 基本數值型別（primitive types，例如 `int`, `double`, `long`, `char`）並不是一種 class，而數值也談不上是個 object。如果要把這一類數值放進容器內，必須將容器元素宣告爲基本型別所對應的外覆類別（wrapper classes），例如圖 3。這實在是非常不方便。JDK1.5

新增自動 boxing（封箱）和 un-boxing（拆箱）特性，也就是在必要時刻自動將數值轉爲外覆物件，或將外覆物件轉爲數值。有了這項特性，我們可以將圖 3 改寫爲圖 4，那就方便多了。

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

圖 3 / 容器元素必須是 object，不可是數值，所以必須使用外覆型別（wrapper）。

```
LinkedList<Integer> iList = new LinkedList<Integer>();
iList.add(0);           //boxing
iList.add(1);
iList.add(5);
iList.add(2);
int i = iList.get(2);   //un-boxing
```

圖 4 / JDK1.5 新增的 boxing/un-boxing 特性，使得以方便地將數值放進容器。

使用 Generic Algorithms

在 Java 程式庫中，針對容器而設計的 algorithms 並不多（不像 C++ 標準程式庫所提供的那麼多），它們都被置於 `java.util.Collections` 內以 static methods 的形式呈現，例如 `sort()`, `max()`, `min()`, `copy()`, `fill()`。圖 5 是兩個運用實例，其語法和 C++完全相同：使用 generic algorithms 時並不需要以角括號（<>）爲「參數化型別」做任何具體指定。這種泛型語法自 JDK1.3+GJ 以來不曾改變過。

```
String str = Collections.max(strList);  //strList 見前例（圖 1）
Collections.sort(strList);
```

圖 5 / 運用 `max()`和 `sort()`

自訂 Generic Classes

先前的 `LinkedList<T>` 運用實例中，我曾假設 `Stroke`, `Rect`, `Circle` 皆繼承自 `Shape`。如果我們希望這些 classes 有足夠的彈性，讓用戶得以在運用這些 classes 時才指定其內部數據（長、寬、半徑等等）的型別，那就得用上泛型語法，如圖 6，

而先前的運用實例也得對應地修改爲圖 7。

```
public abstract class Shape {
  public abstract void draw();
}
```

```
public class Rect<T> extends Shape
                    implements Serializable {
  T m_left, m_top, m_width, m_height;
  public Rect(T left, T top, T width, T height ) { ... }
  ...
}
```

```
public class Circle<T> extends Shape
                     implements Serializable {
  T m_x, m_y, m_r;
  public Circle(T x, T y, T r) { ... }
  ...
}
```

```
public class Stroke<W,T> extends Shape
                       implements Serializable {
  W m_width;
  ArrayList<T> m_ia;
  public Stroke(W width, ArrayList<T> ia) { ... }
  ...
}
```

圖 6 / 自訂 generic classes。本圖實現圖 2a 的繼承體系，並以「參數化型別」（圖中灰色的 `T,W` 等等）代表各 classes 內的數據型別。

```
    LinkedList<Shape> sList = new LinkedList<Shape>();
    sList.add(new Stroke<Integer,Integer>(...));
    sList.add(new Rect<Integer>(...));
    sList.add(new Circle<Integer>(...));
```

圖 7 / 容器的每個元素型別都是 generic classes，所以製造元素時必須使用泛型語法（角括號）。請與圖 2b 比較。

圖 6 和圖 7 的泛型語法自 JDK1.3+GJ 以來不曾改變過。它迥異於 C++，後者要求程式必須在 class 名稱前加上語彙單元 `template<>`，藉此告訴編譯器哪些符號是型別參數（type parameters），如圖 8。

```
template <typename T>
class Rect : public Shape
```

```
{
  private:
     T m_left, m_top, m_width, m_height;
  public:
     Rect(T left, T top, T width, T height ) { ... }
     ...
}
```

圖 8 / C++ class 必須以 `template<typename T>` 這種語彙單元型式，告訴編譯器 `T` 是個參數化型別。請與圖 6 之同名 Java class 比較。

現在讓我們看看 Java 程式庫源碼，從中學習更多的泛型語法。圖 9a 是 `java.util.ArrayList` 的 JDK1.5 源碼，圖 9b 是其 JDK 1.4 源碼，可資比較。

```
#001 public class ArrayList<E> extends AbstractList<E>
#002                          implements List<E>, RandomAccess,
#003                             Cloneable, java.io.Serializable
#004 {
#005     private transient E[] elementData;
#006     private int size;
#007     public ArrayList(int initialCapacity) {
#008          super();
#009          // check if (initialCapacity < 0)...
#010          this.elementData = (E[])new Object[initialCapacity];
#011     }
#012
#013     public ArrayList() {
#014          this(10);
#015     }
#016     ...
#017 }
```

圖 9a / JDK1.5 的 `java.util.ArrayList` 源碼

```
#001 public class ArrayList extends AbstractList
#002                        implements List, RandomAccess,
#003                               Cloneable, java.io.Serializable
#004 {
#005     private transient Object elementData[];
#006     private int size;
#007     public ArrayList(int initialCapacity) {
#008        super();
#009        // check if (initialCapacity < 0) ...
#010        this.elementData = new Object[initialCapacity];
#011     }
#012
```

```
#013     public ArrayList() {
#014        this(10);
#015     }
#016     ...
#017 }
```

圖 9b / JDK1.4 的 java.util.ArrayList 源碼

從圖 9a 可以看出，參數型別（圖中的 E）不但可以繼續被沿用做爲 base class 或 base interfaces 的參數型別，也可以出現在 class 定義區內「具體型別可以出現」的任何地方。不過，例外還是有的，例如這一行：

```
#010 this.elementData = (E[])new Object[initialCapacity];
```

不能寫成：

```
#010 this.elementData = new E[initialCapacity];
```

那會出現 generic array creation error.

目訂 Generic Algorithms

定義於任何 classes 內的任何一個 static method，你都可以說它是個 algorithm。如果這個 method 帶有參數化型別，我們就稱它是 generic algorithm。例如：

```
//在某個 class 之內
public static <T> T gMethod (List<T> list) { ... }
```

這種語法和 generic classes 有相當程度的不同：泛型符號<T> 必須加在 class 名稱之後，卻必須加在 method 名稱（及回傳型別）之前。

JDK 1.5 比以前版本增加了更多彈性，允許所謂 **bounded type parameter**，意指「受到更多約束」的型別參數。下例表示 gMethod()所收到的引數不但必須是個 List，而且其元素型別必須實作 Comparable：

```
public static <T extends Comparable<T>> T gMethod (List<T> list)
{ ... }
```

這種「受到更多約束」的型別參數寫法，雖然不存在於 JDK1.4+JSR14，但其實原本存在於 JDK1.3+GJ 中，只不過用的是另一個關鍵字：

```
public static <T implements Comparable<T>> T gMethod (List<T> list)
```

JDK 1.5 還允許將「不被 method 實際用到」的型別參數以符號 '?' 表示，例如：

```
public static    List<?> gMethod (List<?> list)
{
  return list;    // 本例簡單地原封不動傳回
}
```

此例 gMethod()接受一個 List（無論其元素型別是什麼），傳回一個 List（無論其元素型別是什麼）。由於不存在（或說不在乎）型別參數（因爲 method 內根本不去用它），也就不必如平常一般在回傳型別之前寫出<T>來告知編譯器了。

上面這個例子無法真正表現出符號 '?' 的用途。真正的好例子請看 JDK1.5 的 java.util.Collections 源碼，見圖 10a。圖 10b 則是其 JDK1.4 源碼，可資比較。請注意，例中的 '?' 不能被替換爲任何其他符號。圖 10a 程式碼所描述的意義，請見圖 11 的細部解釋。

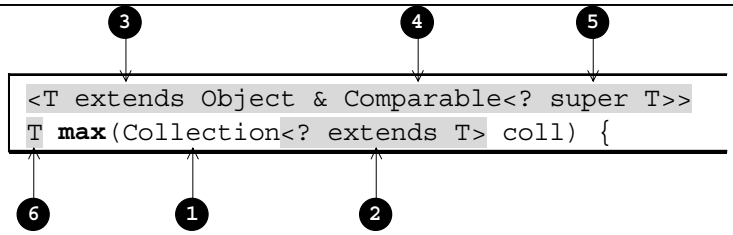
```
#001 public class Collections
#002 ...
#003   public static
#004   <T extends Object & Comparable<? super T>>
#005   T max(Collection<? extends T> coll) {
#006      Iterator<? extends T> i = coll.iterator();
#007      T candidate = i.next();
#008
#009      while(i.hasNext()) {
#010         T next = i.next();
#011         if (next.compareTo(candidate) > 0)
#012              candidate = next;
#013      }
#014      return candidate;
#015   }
#016   ...
#017 } // of Collections
```

圖 10a / JDK1.5 的 java.util.Collections 源碼。

```
#001 public class Collections
#002 ...
#003    public static
#004                          //這裡我刻意放空一行，以利與 JDK1.5 源碼比較
```

```
#005     Object max(Collection coll) {
#006    Iterator i = coll.iterator();
#007    Comparable candidate = (Comparable)(i.next());
#008
#009        while(i.hasNext()) {
#010           Comparable next = (Comparable)(i.next());
#011           if (next.compareTo(candidate) > 0)
#012                candidate = next;
#013        }
#014        return candidate;
#015     }
#016 ...
#017 } // of Collections
```

圖 10b / JDK1.4 的 `java.util.Collections` 源碼。



1. `max()` 接收一個 `Collection` object。
2. 該 `Collection` object 所含元素必須是 `T`-derived object。
3. `T` 必須繼承 `Object`（這倒不必明說，因爲 Java 必定如此）。
4. `T` 必須實作 `Comparable`。
5. `Comparable` 所比較的型別必須是 T 的 super type。
6. `max()` 傳回 `T` object。

圖 11 / 本圖詳細說明圖 10a 的怪異內容（#4, #5 兩行）

面對圖 11 如此「怪異而罕見」的語法，給個實際用例就清楚多了：

```
LinkedList<Shape> sList = new LinkedList<Shape>();
...
Shape s = Collections.max(sList);
```

我們讓 `Collections.max()` 接受一個先前曾經說過的 `LinkedList<Shape>`（見圖 2a,b），那是個 `Collections` object（符合圖 11 條件 1），其中每個元素都是 `Shape`-derived objects（符合圖 11 條件 2），因此本例中的 `T` 就是 `Shape`。`Shape` 的確繼承自 `Object`（符合圖 11 條件 3），並且必須實作 `Comparable`（才能符合圖 11

條件 4），而被比較物的型別必須是 Shape 的 super class（才能符合圖 11 條件 5）。max()比較所得之最大值以 Shape 表示（符合圖 11 條件 6）——這是合理的，因爲不知道比較出來的結果會是 Rect 或 Circle 或 Stroke，但無論如何它們都可以向上轉型爲 Shape。

爲了完成上述的條件 4 和條件 5，先前的 Shape 必須修改，使得以被比較。也就是說 Shape 必須實作 Comparable 介面，如圖 12，其中針對 compareTo()用上了典型的 **Template Method** 設計範式（design pattern），再令每一個 Shape-derived classes 都實作 L()如圖 13，這就大功告成了。

```
public abstract class Shape implements Comparable<Shape>  {
  ...
  public abstract double L();         //計算周長
  public int compareTo(Shape o) {     //假設「以周長爲比較依據」合理!
    return (this.L() < o.L() ? -1 : (this.L() == o.L() ? 0 : 1));
  }
}
```

圖 12 / 修改圖 7 的 Shape class

```
  public double L() {
    ... 計算周長     //這裡有點學問，見最後「擦拭法帶來的遺憾」討論。
    return xxx;     //傳回周長
  }
```

圖 13 / 每個 Shape-derived classes 都必須實作出如此型式的 L()。

參數化型別（Parameterized type）存在多久？

究竟 generic classes 所帶的參數化型別，在編譯後是否還保留？或者說，假設我們將一個元素型別爲 Integer 的 LinkedList 容器寫入檔案，而後讀出並恢復「先前被 serialized（序列化）至檔案」的容器，如圖 14。此時我們的第一個考慮作法或許如下（和最初的宣告完全相同）：

```
LinkedList<Integer> iList2 = (LinkedList<Integer>)in.readObject();
```

但 JDK1.5 編譯器發出警告，告訴我們 "unchecked cast"（JDK1.4 則直接抱怨它是錯誤的）。改成這樣情況亦同：

```
LinkedList iList2 = (LinkedList<Integer>)in.readObject();
```

看來編譯器面對即將被 deSerialized（反序列化）讀得的型別資訊，似乎無法判別是否可以成功轉型爲 `LinkedList<Integer>`。另一種寫法是：

```
LinkedList<Integer> iList2 = (LinkedList)in.readObject();
```

這一次 JDK1.5 編譯器發出的警告訊息是："unchecked conversion"。改爲這樣更非警告可以善了：

```
LinkedList<Integer> iList2 = in.readObject();
```

對此 JDK1.5 編譯器會直接報錯："incompatible types"。如果改成這樣：

```
LinkedList iList2 = (LinkedList)in.readObject();
```

這才是既無錯誤又無警告的完美寫法。

```
LinkedList<Integer> iList = new LinkedList<Integer>();
...
ObjectOutputStream out =
    new ObjectOutputStream(
      new FileOutputStream("out"));

out.writeObject(iList);    //寫入
out.close();

ObjectInputStream in =
  new ObjectInputStream(
    new FileInputStream("out"));

//...這裡準備進行讀取動作 readObject()
```

圖 14 / 將元素型別爲 `Integer` 的容器寫入檔案，而後準備讀出。

由以上測試結果可以預期，似乎存在這一事實：當 object 被寫入檔案，即失去其泛型型別參數（如果有的話）。因此讀回的只是「非泛型」的 class 資訊。如果上述最後那個「完美」寫法改成這樣：

```
ArrayList iList2 = (ArrayList)in.readObject();
```

仍可順利編譯，在編譯器的眼裡看來也很完美，但執行期會出現異常，告知「讀入的 class 資訊」和「程式預備接收的 class 資訊」不相符合。異常訊息如下：

```
Exception in thread "main" java.lang.ClassCastException:
```

```
java.util.LinkedList
```

我們可以觀察 serialization 的輸出檔獲得證據。從圖 15 可以看出來，檔案內記錄的class名稱是`java.util.LinkedList`，並一併記錄了元素型別`java.lang.Integer`（及其 base class `java.lang.Number`）。但元素型別其實是針對每一個元素都會記錄的（當然啦，如果遇上相同型別的元素，這些資訊並不會又被傻傻地完整記錄一遍，那太浪費時間和空間，而是只記錄一個 handle，詳見《Java 的物件永續之道》，網址列於文後）。這些記錄對於 deSerialization 過程中恢復容器原型和內容有其必要，但無法讓編譯器推論當初使用的是 `LinkedList` 容器或是`LinkedList<Integer>`容器。

```
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 04 73 J`."...xpw.....s
000030: 72 00 11 6A 61 76 61 2E  6C 61 6E 67 2E 49 6E 74 r..java.lang.Int
000040: 65 67 65 72 12 E2 A0 A4  F7 81 87 38 02 00 01 49 eger.?艾?8...I
000050: 00 05 76 61 6C 75 65 78  72 00 10 6A 61 76 61 2E ..valuexr..java.
000060: 6C 61 6E 67 2E 4E 75 6D  62 65 72 86 AC 95 1D 0B lang.Number  ...
```

圖 15 / 將元素型別爲 `Integer` 的容器寫入檔案，而後準備讀出。

Java 擦拭法 vs. C++膨脹法

爲什麼 Java 容器的參數化型別無法永續存在於檔案（或其他資料流）內？本文一開始已經說過，這些容器被設計用來存放 `Object`-derived 元素，而 Java 擁有單根繼承體系，所有 Java classes 都繼承自 `java.lang.Object`，因此任何 Java objects 都可以被放進各種容器，換句話說 Java 容器本來就是一種「泛型」的異質容器。今天加上參數化型別反而是把它「窄化」了。「泛型」之於 Java，只是一個角括號面具（當然這個面具帶給了程式開發過程某些好處）；摘下面具，原貌即足夠應付一切。因此 Java 使用所謂「擦拭法」來對待角括號內的參數化型別，如圖 16a。下面是「擦拭法」的四大要點（另有其他枝節，本文不談）：

- 一個參數化型別經過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經過擦拭後應該獲得型別本身（於是 *Byte* 被擦拭成爲 *Byte*）
- 一個型別參數經過擦拭後的結果爲 *Object*（於是 *T* 被擦拭後變成 *Object*）

- 如果某個 method call 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。

這種觀念和 C++的泛型容器完全不同。C++容器是以同一套程式碼，由編譯器根據其被使用時所被指定的「不同的參數化型別」建立出不同版本。換句話說一份 template（範本、模板）被膨脹爲多份程式碼，如圖 16b。

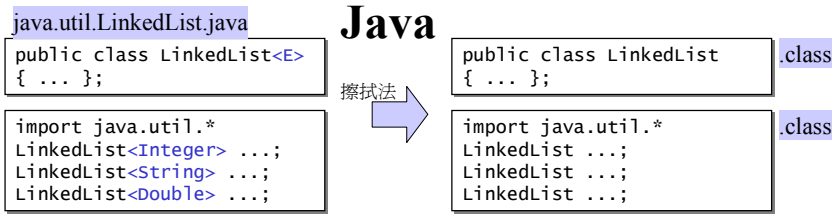


圖 16a / Java 以擦拭法成就泛型

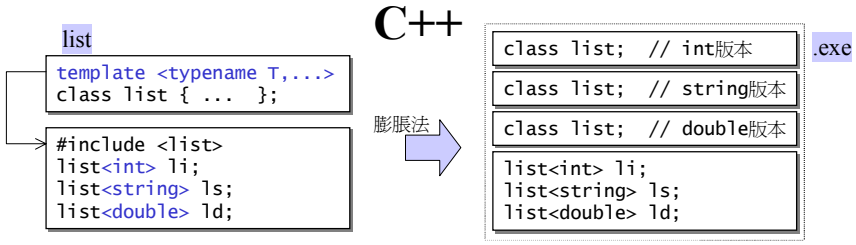


圖 16b / C++以膨脹法成就泛型

擦拭法帶來的遺憾

先前談到的 "Shape" 多型實例（圖 2a），其中的 class `Rect`：

```
public class Rect<T> extends Shape
                    implements Serializable {
  T m_left, m_top, m_width, m_height;
  public Rect(T left, T top, T width, T height ) { ... }
  ...
}
```

經過擦拭後變成了：

```
public class Rect extends Shape
```

```
                    implements Serializable {
  Object m_left, m_top, m_width, m_height;
  public Rect(Object left, Object top, Object width, Object height )
{ ... }
  ...
}
```

這麼一來，任何數值運算，例如先前提過的「周長計算」L()將無法編譯，如圖 17：

```
//class Rect<T> 內
public double L() {
   return (double)((m_width + m_height) * 2);
}
```

圖 17 / L()發生錯誤

錯誤訊息是："operator + cannot be applied to T,T"。是的，兩個 Object object 如何相加呢？Java 並沒有提供像 C++ 那樣的運算子重載（operator overloading）功能！可以說，圖 2a 的 Shape 繼承體系只是介面正確，一旦面臨某些情況，卻無實用性。我的結論是，將參數化型別用於 Java non-collection classes 身上，恐怕會面臨許多束縛。（註：讀者來函提供了此問題的一個解答，見本文末尾添加之補充）

更多資訊

以下是與本文主題相關的更多討論。這些資訊可以彌補本文篇幅限制而造成的不足，並帶給您更多視野。

- 《Java 泛型技術之發展》,by 侯捷。http://www.jjhou.com/javatwo-2002-generics-in-jdk14.pdf
- 《Java 的物件永續之道》,by 侯捷。http://www.jjhou.com/javatwo-2003-serialization-doc.pdf

■補充

讀者 AutoWay 針對無法計算周長這個問題，來信如下：

> From: AutoWay
>
> Sent: Monday, January 17, 2005 7:57 PM
>
> Subject: 《JDK 1.5 的泛型實現》讀者回應
>
> 侯捷兄：隨函寄上 Rect.java 程式之修訂，俾可以進行「周長計算」。修訂內容如下：在設定 type parameter 時，同時宣告其 type bound，例如本例修改爲 `<T extends Number>`。系統進行編譯時，就知道 `T` 是 `Number` 或其 subclass；因此內部程式就可運用 `Number` 提供的 methods 進行運算了。
>
> 我想，這就是 type parameters 之所以提供 type bounds 機制的主要原因。如果 type bounds 光用來限制 type arguments 之傳遞，實在沒啥意思！感謝本文揭示的例子，讓我對 type bounds 有進一步的省思與認識；若有謬誤，亦請來信指教。

侯捷回覆：非常感謝 AutoWay 兄的指正，解除了我的盲點。整理於下。圖 6 之程式碼應改爲：

```
public class Rect<T extends Number> extends Shape
                    implements Serializable {
  ...
}
```

```
public class Circle<T extends Number> extends Shape
                      implements Serializable {
  ...
}
```

```
public class Stroke<W extends Number,T extends Number> extends Shape
                         implements Serializable {
  ...
}
```

圖 17 程式碼應改爲：

```
//class Rect<T extends Number> 內
public double L() {
   return (m_width.doubleValue() + m_height.doubleValue()) * 2;
}
```

另兩個 classes（`Circle` 和 `Stroke`）同理修改。

侯捷觀點