# java 泛型

package no.jinni.genericdemo;

public class GenericDemo\_01 {

public static void main(String args[]) {

Point p = new Point();

p.setX(11);

p.setY(20);

int x = (Integer) p.getX();

int y = (Integer) p.getY();

System.out.println("x:" + x);

System.out.println("y:" + y);

}

}

class Point {

private Object x;

private Object y;

public Object getX() {

return x;

}

public void setX(Object x) {

this.x = x;

}

public Object getY() {

return y;

}

public void setY(Object y) {

this.y = y;

}

}

package no.jinni.genericdemo;

public class GenericDemo\_02 {

public static void main(String args[]) {

Point p = new Point();

p.setX("22");

p.setY("20");

String x = (String) p.getX();

String y = (String) p.getY();

System.out.println("x:" + x);

System.out.println("y:" + y);

}

}

class Point {

private Object x;

private Object y;

public Object getX() {

return x;

}

public void setX(Object x) {

this.x = x;

}

public Object getY() {

return y;

}

public void setY(Object y) {

this.y = y;

}

}

package no.jinni.genericdemo;

public class GenericDemo\_03 {

public static void main(String args[]) {

Point p = new Point();

p.setX("33");

p.setY(20);

String x = (String) p.getX();

String y = (String) p.getY();

System.out.println("x:" + x);

System.out.println("y:" + y);

}

}

class Point {

private Object x;

private Object y;

public Object getX() {

return x;

}

public void setX(Object x) {

this.x = x;

}

public Object getY() {

return y;

}

public void setY(Object y) {

this.y = y;

}

}

/\*

\* 使用泛型，以String建構物件

\*/

package no.jinni.genericdemo;

public class GenericDemo\_04 {

public static void main(String args[]) {

Point<String> p = new Point<String>();

p.setX("44");

p.setY("20");

String x = (String) p.getX();

String y = (String) p.getY();

System.out.println("x:" + x);

System.out.println("y:" + y);

}

}

class Point<A> {

private A x;

private A y;

public A getX() {

return x;

}

public void setX(A x) {

this.x = x;

}

public A getY() {

return y;

}

public void setY(A y) {

this.y = y;

}

}

/\*

\* 以泛型宣告，未指明型態，則以Object型態建構物件

\*/

package no.jinni.genericdemo;

public class GenericDemo\_05 {

public static void main(String args[]) {

Point p = new Point();

p.setX(55);

p.setY(20);

int x = (Integer) p.getX();

int y = (Integer) p.getY();

System.out.println("x:" + x);

System.out.println("y:" + y);

}

}

class Point<B> {

private B x;

private B y;

public B getX() {

return x;

}

public void setX(B x) {

this.x = x;

}

public B getY() {

return y;

}

public void setY(B y) {

this.y = y;

}

}

package no.jinni.genericdemo;

public class GenericDemo\_06 {

public static void main(String args[]) {

Point<Integer> p = new Point<Integer>();

p.setX(66);

find(p);

}

public static void find(Point<Integer> temp) {

temp.setX(200);

System.out.println("x:" + temp.getX());

}

}

class Point<T> {

private T x;

public T getX() {

return x;

}

public void setX(T x) {

this.x = x;

}

}

package no.jinni.genericdemo;

public class GenericDemo\_06 {

public static void main(String args[]) {

Point<String> p = new Point<String>();

p.setX("77");

find(p);

}

public static void find(Point<Integer> temp) {

temp.setX(200);

System.out.println("x:" + temp.getX());

}

/\* 重載

\* public static void find(Point<Integer> temp) {

temp.setX("200");

System.out.println("x:" + temp.getX());

}\*/

}

class Point<T> {

private T x;

public T getX() {

return x;

}

public void setX(T x) {

this.x = x;

}

}

package no.jinni.genericdemo;

public class GenericDemo\_08 {

public static void main(String args[]) {

Point<String> p = new Point<String>();

p.setX("88");

find(p);

}

public static void find(Point<?> temp) {

// temp.setX(200);//不能修改

System.out.println("x:" + temp.getX());

}

}

class Point<T> {

private T x;

public T getX() {

return x;

}

public void setX(T x) {

this.x = x;

}

}

package no.jinni.genericdemo;

public class GenericDemo\_09 {

public static void main(String args[]) {

Point<Integer> p = new Point<Integer>();

p.setX(99);

find(p);

}

public static void find(Point<? extends Number> temp) {

// temp.setX(200);//不能修改

System.out.println("x:" + temp.getX());

}

}

class Point<T> {

private T x;

public T getX() {

return x;

}

public void setX(T x) {

this.x = x;

}

}

# java 泛型

2016年12月30日 11:44:閱讀數：1113對java的泛型特性的瞭解僅限於表面的淺淺一層，直到在學習設計模式時發現有不瞭解的用法，才想起詳細的記錄一下。

本文參考[java 泛型詳解](http://www.cnblogs.com/sunwei2012/archive/2010/10/08/1845938.html)、[Java中的泛型方法](http://www.cnblogs.com/iyangyuan/archive/2013/04/09/3011274.html)、 [java泛型詳解](http://blog.csdn.net/caihuangshi/article/details/51278793)

## 概述

泛型在java中有很重要的地位，在物件導向程式設計及各種設計模式中有非常廣泛的應用。

什麼是泛型？為什麼要使用泛型？

泛型，即”參數化型態”。一提到參數(Parameter)，最熟悉的就是定義方法時有參數，然後存取此方法時傳遞引數(Argument)。那麼參數化型態怎麼理解呢？顧名思義，就是將型態由原來的具體的型態參數化，類似於方法中的變數參數，此時型態也定義成參數形式(可以稱之為型態參數)，然後在使用/存取時傳入具體的型態(型態引數)。

泛型的本質是為了參數化型態(在不創建新的型態的情況下，通過泛型指定的不同型態來控制參數具體限制的型態)。也就是說在泛型使用過程中，操作的資料型態被指定為一個參數，這種參數型態可以用在類別、介面和方法中，分別被稱為泛型類別、泛型介面、泛型方法。

## 範例

List arrayList = new ArrayList();

arrayList.add("aaaa");

arrayList.add(100);

for(int i = 0; i< arrayList.size();i++){

String item = (String)arrayList.get(i);

Log.d("泛型測試","item = " + item);

}

毫無疑問，程式的執行結果會以崩潰結束：

java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String

arrayList可以存放任意型態，例子中添加一個String型態，添加一個Integer型態，再使用時都以String的方式使用，因此程式崩潰了。為了解決類似這樣的問題(在編譯階段就可以解決)，泛型應運而生。

將第一行宣告初始化list的程式碼更改一下，編譯器會在編譯階段就能夠幫我們發現類似這樣的問題。

List<String> arrayList = new ArrayList<String>();

...

//arrayList.add(100); 在編譯階段，編譯器就會報錯

## 特性

泛型只在編譯階段有效。看下面的程式碼：

List<String> stringArrayList = new ArrayList<String>();

List<Integer> integerArrayList = new ArrayList<Integer>();

Class classStringArrayList = stringArrayList.getClass();

Class classIntegerArrayList = integerArrayList.getClass();

if(classStringArrayList.equals(classIntegerArrayList)){

Log.d("泛型測試","型態相同");

}

輸出結果：D/泛型測試: 型態相同。

通過上面的例子可以證明，在編譯之後程式會採取去泛型化的措施。也就是說Java中的泛型，只在編譯階段有效。在編譯過程中，正確檢驗泛型結果後，會將泛型的相關資訊擦出，並且在物件進入和離開方法的邊界處添加型態檢查和型態轉換的方法。也就是說，泛型資訊不會進入到執行時階段。

對此總結成一句話：泛型型態在邏輯上可以看成是多個不同的型態，實際上都是相同的基本型態。

## 泛型的使用

泛型有三種使用方式，分別為：泛型類別、泛型介面、泛型方法。

### 泛型類別

泛型型態用於類別的定義中，被稱為泛型類別。通過泛型可以完成對一組類別的操作對外開放相同的介面。最典型的就是各種容器類別，如：List、Set、Map。

泛型類別的最基本寫法：

class 類別名稱 <泛型標識：可以隨便寫任意標識號，標識指定的泛型的型態>{

private 泛型標識/\*(成員變數型態)\*/ var;

...

}

}

一個最普通的泛型類別：

//此處T可以隨便寫為任意標識，常見的如T、E、K、V等形式的參數常用於表示泛型

//在產生實體泛型類別時，必須指定T的具體型態

public class Generic<T>{

//key這個成員變數的型態為T, T的型態由外部指定

private T key;

public Generic(T key){//泛型構造方法參數key的型態也為T，T的型態由外部指定

this.key = key;

}

public T getKey(){//泛型方法getKey的返回數值型態為T，T的型態由外部指定

return key;

}

}

//泛型的型態參數只能是類別型態(包括自訂類別)，不能是基本型態

//傳入的引數型態需與泛型的型態參數型態相同，即為Integer.

Generic<Integer> genericInteger = new Generic<Integer>(123456);

//傳入的引數型態需與泛型的型態參數型態相同，即為String.

Generic<String> genericString = new Generic<String>("key\_vlaue");

Log.d("泛型測試","key is " + genericInteger.getKey());

Log.d("泛型測試","key is " + genericString.getKey());

12-27 09:20:0432 13063-13063/? D/泛型測試: key is 123412-27 09:20:0432 13063-13063/? D/泛型測試: key is key\_vlaue

定義的泛型類別，就一定要傳入泛型型態引數麼？並不是這樣，在使用泛型的時候如果傳入泛型引數，則會根據傳入的泛型引數做相應的限制，此時泛型才會產生本應產生的限制作用。如果不傳入泛型型態引數的話，在泛型類別中使用泛型的方法或成員變數定義的型態可以為任何的型態。

看一個例子：

Generic generic = new Generic("111111");

Generic generic1 = new Generic(4444);

Generic generic2 = new Generic(555);

Generic generic3 = new Generic(false);

Log.d("泛型測試","key is " + generic.getKey());

Log.d("泛型測試","key is " + genericgetKey());

Log.d("泛型測試","key is " + genericgetKey());

Log.d("泛型測試","key is " + genericgetKey());

D/泛型測試: key is 1111D/泛型測試: key is 44D/泛型測試: key is 5D/泛型測試: key is false

注意：

泛型的型態參數只能是類別型態，不能是基本型態。

不能對確切的泛型型態使用instanceof操作。如下麵的操作是非法的，編譯時會出錯。

if(ex\_num instanceof Generic<Number>){

}

### 泛型介面

泛型介面與泛型類別的定義及使用基本相同。泛型介面常被用在各種類別的生產器中，可以看一個例子：

//定義一個泛型介面

public interface Generator<T>{

public T next();

}

當實現泛型介面的類別，未傳入泛型引數時：

/\*\*

\*未傳入泛型引數時，與泛型類別的定義相同，在宣告類別的時候，需將泛型的宣告也一起加到類別中

\*即：class FruitGenerator<T> implements Generator<T>{

\*如果不宣告泛型，如：class FruitGenerator implements Generator<T>，編譯器會報錯："Unknown class" \*/

class FruitGenerator<T> implements Generator<T>{

@Override

public T next(){

return null;

}

}

當實現泛型介面的類別，傳入泛型引數時：

/\*\*

\*傳入泛型引數時：

\*定義一個生產器實現這個介面,雖然我們只創建了一個泛型介面Generator<T>

\*但是我們可以為T傳入無數個引數，形成無數種型態的Generator介面。

\*在實現類別實現泛型介面時，如已將泛型型態傳入引數型態，則所有使用泛型的地方都要替換成傳入的引數型態

\*即：Generator<T>，public T next();中的的T都要替換成傳入的String型態。 \*/

public class FruitGenerator implements Generator<String>{

private String[] fruits = new String[]{"Apple", "Banana", "Pear"};

@Override

public String next(){

Random rand = new Random();

return fruits[rand.nextInt(3)];

}

}

### 泛型萬用字元

我們知道Ingeter是Number的一個子類別，同時在特性章節中我們也驗證過Generic<Ingeter>與Generic<Number>實際上是相同的一種基本型態。那麼問題來了，在使用Generic<Number>作為參數的方法中，能否使用Generic<Ingeter>的實例傳入呢？在邏輯上類似於Generic<Number>和Generic<Ingeter>是否可以看成具有父子關係的泛型型態呢？

為了弄清楚這個問題，我們使用Generic<T>這個泛型類別繼續看下面的例子：

public void showKeyValue1(Generic<Number> obj){

Log.d("泛型測試","key value is " + obj.getKey());

}

Generic<Integer> gInteger = new Generic<Integer>(123);

Generic<Number> gNumber = new Generic<Number>(456);

showKeyValue(gNumber);

//showKeyValue這個方法編譯器會為我們報錯：Generic<java.lang.Integer>

//cannot be applied to Generic<java.lang.Number>

//showKeyValue(gInteger);

通過提示資訊我們可以看到Generic<Integer>不能被看作為`Generic<Number>的子類別。由此可以看出:同一種泛型可以對應多個版本(因為參數型態是不確定的)，不同版本的泛型類別實例是不相容的。

回到上面的例子，如何解決上面的問題？總不能為了定義一個新的方法來處理Generic<Integer>型態的類別，這顯然與java中的多台理念相違背。因此我們需要一個在邏輯上可以表示同時是Generic<Integer>和Generic<Number>父類別的參考型態。由此型態萬用字元應運而生。

我們可以將上面的方法改一下：

public void showKeyValue1(Generic<?> obj){

Log.d("泛型測試","key value is " + obj.getKey());

}

型態萬用字元一般是使用？代替具體的型態引數，注意了，此處’？’是型態引數，而不是型態參數 。重要說三遍！此處’？’是型態引數，而不是型態參數 ！ 此處’？’是型態引數，而不是型態參數 ！再直白點的意思就是，此處的？和Number、String、Integer一樣都是一種實際的型態，可以把？看成所有型態的父類別。是一種真實的型態。

可以解決當具體型態不確定的時候，這個萬用字元就是 ? ；當操作型態時，不需要使用型態的具體功能時，只使用Object類別中的功能。那麼可以用 ? 萬用字元來表未知型態。

### 泛型方法

在java中,泛型類別的定義非常簡單，但是泛型方法就比較複雜了。

尤其是我們見到的大多數泛型類別中的成員方法也都使用了泛型，有的甚至泛型類別中也包含著泛型方法，這樣在初學者中非常容易將泛型方法理解錯了。

泛型類別，是在產生實體類別的時候指明泛型的具體型態；泛型方法，是在存取方法的時候指明泛型的具體型態 。

/\*\*

\*泛型方法的基本介紹

\*@param tClass 傳入的泛型引數

\*@return T 返回值為T型態

\*說明：

\*1)public 與 返回值中間<T>非常重要，可以理解為宣告此方法為泛型方法。

\*2)只有宣告了<T>的方法才是泛型方法，泛型類別中的使用了泛型的成員方法並不是泛型方法。

\*3)<T>表明該方法將使用泛型型態T，此時才可以在方法中使用泛型型態T。

\*4)與泛型類別的定義一樣，此處T可以隨便寫為任意標識，常見的如T、E、K、V等形式的參數常用於表示泛型。 \*/

public <T> T genericMethod(Class<T> tClass)throws InstantiationException ,

IllegalAccessException{

T instance = tClass.newInstance();

return instance;

}

Object obj = genericMethod(Class.forName("com.test.test"));

#### 泛型方法的基本用法

光看上面的例子有的同學可能依然會非常迷糊，我們再通過一個例子，把我泛型方法再總結一下。

public class GenericTest{

//這個類別是個泛型類別，在上面已經介紹過

public class Generic<T>{

private T key;

public Generic(T key){

this.key = key;

}

//我想說的其實是這個，雖然在方法中使用了泛型，但是這並不是一個泛型方法。

//這只是類別中一個普通的成員方法，只不過他的返回值是在宣告泛型類別已經宣告過的泛型。

//所以在這個方法中才可以繼續使用 T 這個泛型。

public T getKey(){

return key;

}

/\*\*

\* 這個方法顯然是有問題的，在編譯器會給我們提示這樣的錯誤資訊"cannot reslove symbol E"

\* 因為在類別的宣告中並未宣告泛型E，所以在使用E做參數和返回數值型態時，編譯器會無法識別。

public E setKey(E key){

this.key = keu

}

\*/

}

/\*\*

\* 這才是一個真正的泛型方法。

\* 首先在public與返回值之間的<T>必不可少，這表明這是一個泛型方法，並且宣告了一個泛型T

\* 這個T可以出現在這個泛型方法的任意位置.

\* 泛型的數量也可以為任意多個

\* 如：public <T,K> K showKeyName(Generic<T> container){

\* ...

\* } \*/

public <T> T showKeyName(Generic<T> container){

System.out.println("container key :" + container.getKey());

//當然這個例子舉的不太合適，只是為了說明泛型方法的特性。

T test = container.getKey();

return test;

}

//這也不是一個泛型方法，這就是一個普通的方法，只是使用了Generic<Number>這個泛型類別做參數而已。

public void showKeyValue1(Generic<Number> obj){

Log.d("泛型測試","key value is " + obj.getKey());

}

//這也不是一個泛型方法，這也是一個普通的方法，只不過使用了泛型萬用字元?

//同時這也印證了泛型萬用字元章節所描述的，?是一種型態引數，可以看做為Number等所有類別的父類別

public void showKeyValue2(Generic<?> obj){

Log.d("泛型測試","key value is " + obj.getKey());

}

/\*\*

\* 這個方法是有問題的，編譯器會為我們提示錯誤資訊："UnKnown class 'E' "

\* 雖然我們宣告了<T>,也表明了這是一個可以處理泛型的型態的泛型方法。

\* 但是只宣告了泛型型態T，並未宣告泛型型態E，因此編譯器並不知道該如何處理E這個型態。

public <T> T showKeyName(Generic<E> container){

...

} \*/

/\*\*

\* 這個方法也是有問題的，編譯器會為我們提示錯誤資訊："UnKnown class 'T' "

\* 對於編譯器來說T這個型態並未項目中宣告過，因此編譯也不知道該如何編譯這個類別。

\* 所以這也不是一個正確的泛型方法宣告。

public void showkey(T genericObj){

} \*/

public static void main(String[] args){

}

}

#### 類別中的泛型方法

當然這並不是泛型方法的全部，泛型方法可以出現雜任何地方和任何場景中使用。但是有一種情況是非常特殊的，當泛型方法出現在泛型類別中時，我們再通過一個例子看一下

public class GenericFruit{

class Fruit{

@Override

public String toString(){

return "fruit";

}

}

class Apple extends Fruit{

@Override

public String toString(){

return "apple";

}

}

class Person{

@Override

public String toString(){

return "Person";

}

}

class GenerateTest<T>{

public void show\_1(T t){

System.out.println(t.toString());

}

//在泛型類別中宣告了一個泛型方法，使用泛型E，這種泛型E可以為任意型態。可以型態與T相同，也可以不同。

//由於泛型方法在宣告的時候會宣告泛型<E>，因此即使在泛型類別中並未宣告泛型，編譯器也能夠正確識別泛型方法中識別的泛型。

public <E> void show\_3(E t){

System.out.println(t.toString());

}

//在泛型類別中宣告了一個泛型方法，使用泛型T，注意這個T是一種全新的型態，可以與泛型類別中宣告的T不是同一種型態。

public <T> void show\_2(T t){

System.out.println(t.toString());

}

}

public static void main(String[] args){

Apple apple = new Apple();

Person person = new Person();

GenerateTest<Fruit> generateTest = new GenerateTest<Fruit>();

//apple是Fruit的子類別，所以這裡可以

generateTest.show\_1(apple);

//編譯器會報錯，因為泛型型態引數指定的是Fruit，而傳入的引數類別是Person

//generateTest.show\_1(person);

//使用這兩個方法都可以成功

generateTest.show\_2(apple);

generateTest.show\_2(person);

//使用這兩個方法也都可以成功

generateTest.show\_3(apple);

generateTest.show\_3(person);

}

}

#### 泛型方法與可變參數

再看一個泛型方法和可變參數的例子：

public <T> void printMsg(T... args){

for(T t : args){

Log.d("泛型測試","t is " + t);

}

}

printMsg("111",222,"aaaa","2324",555);

#### 靜態方法與泛型

靜態方法有一種情況需要注意一下，那就是在類別中的靜態方法使用泛型：靜態方法無法訪問類別上定義的泛型；如果靜態方法操作的引用資料型態不確定的時候，必須要將泛型定義在方法上。

即：如果靜態方法要使用泛型的話，必須將靜態方法也定義成泛型方法 。

public class StaticGenerator<T>{

....

....

/\*\*

\* 如果在類別中定義使用泛型的靜態方法，需要添加額外的泛型宣告(將這個方法定義成泛型方法)

\* 即使靜態方法要使用泛型類別中已經宣告過的泛型也不可以。

\* 如：public static void show(T t){..},此時編譯器會提示錯誤資訊：

"StaticGenerator cannot be refrenced from static context"

\*/

public static <T> void show(T t){

}

}

#### 泛型方法總結

泛型方法能使方法獨立於類別而產生變化，以下是一個基本的指導原則：

無論何時，如果你能做到，你就該儘量使用泛型方法。也就是說，如果使用泛型方法將整個類別泛型化，那麼就應該使用泛型方法。另外對於一個static的方法而已，無法訪問泛型型態的參數。所以如果static方法要使用泛型能力，就必須使其成為泛型方法。

## 泛型上下邊界

在使用泛型的時候，我們還可以為傳入的泛型型態引數進行上下邊界的限制，如：型態引數只准傳入某種型態的父類別或某種型態的子類別。

為泛型添加上邊界，即傳入的型態引數必須是指定型態的子型態

public void showKeyValue1(Generic<? extends Number> obj){

Log.d("泛型測試","key value is " + obj.getKey());

}

Generic<String> generic1 = new Generic<String>("11111");

Generic<Integer> generic2 = new Generic<Integer>(2222);

Generic<Float> generic3 = new Generic<Float>(4f);

Generic<Double> generic4 = new Generic<Double>(56);

//這一行程式碼編譯器會提示錯誤，因為String型態並不是Number型態的子類別

//showKeyValue1(generic1);

showKeyValue1(generic2);

showKeyValue1(generic3);

showKeyValue1(generic4);

如果我們把泛型類別的定義也改一下:

public class Generic<T extends Number>{

private T key;

public Generic(T key){

this.key = key;

}

public T getKey(){

return key;

}

}

//這一行程式碼也會報錯，因為String不是Number的子類別

Generic<String> generic1 = new Generic<String>("11111");

再來一個泛型方法的例子：

//在泛型方法中添加上下邊界限制的時候，必須在許可權宣告與返回值之間的<T>上添加上下邊界，即在泛型宣告的時候添加

//public <T> T showKeyName(Generic<T extends Number> container)，編譯器會報錯："Unexpected bound"

public <T extends Number> T showKeyName(Generic<T> container){

System.out.println("container key :" + container.getKey());

T test = container.getKey();

return test;

}

通過上面的兩個例子可以看出：泛型的上下邊界添加，必須與泛型的宣告在一起 。

### 關於泛型陣列要提一下

看到了很多文章中都會提起泛型陣列，經過查看sun的說明文檔，在java中是”不能創建一個確切的泛型型態的陣列”的。

也就是說下面的這個例子是不可以的：

List<String>[] ls = new ArrayList<String>[10];

而使用萬用字元創建泛型陣列是可以的，如下面這個例子：

List<?>[] ls = new ArrayList<?>[10];

這樣也是可以的：

List<String>[] ls = new ArrayList[10];

下面使用[Sun](http://docs.oracle.com/javase/tutorial/extra/generics/fineprint.html" \t "_blank)[的一篇文檔](http://docs.oracle.com/javase/tutorial/extra/generics/fineprint.html)的一個例子來說明這個問題：

List<String>[] lsa = new List<String>[10];//Not really allowed.

Object o = lsa;

Object[] oa = (Object[]) o;

List<Integer> li = new ArrayList<Integer>();

li.add(new Integer(3));

oa[1] = li;//Unsound, but passes run time store check

String s = lsa[1].get(0);//Run-time error: ClassCastException.

這種情況下，由於JVM泛型的擦除機制，在執行時JVM是不知道泛型資訊的，所以可以給oa[1]賦上一個ArrayList而不會出現異常，但是在取出資料的時候卻要做一次型態轉換，所以就會出現ClassCastException，如果可以進行泛型陣列的宣告，上面說的這種情況在編譯期將不會出現任何的警告和錯誤，只有在執行時才會出錯。

而對泛型陣列的宣告進行限制，對於這樣的情況，可以在編譯期提示程式碼有型態安全問題，比沒有任何提示要強很多。

下面採用萬用字元的方式是被允許的:陣列的型態不可以是型態變數，除非是採用萬用字元的方式，因為對於萬用字元的方式，最後取出資料要做強制型態轉換。

List<?>[] lsa = new List<?>[10];//OK, array of unbounded wildcard type.

Object o = lsa;

Object[] oa = (Object[]) o;

List<Integer> li = new ArrayList<Integer>();

li.add(new Integer(3));

oa[1] = li;//Correct.

Integer i = (Integer) lsa[1].get(0);//OK

## 最後

本文中的例子主要是為了闡述泛型中的一些思想而簡單舉出的，並不一定有著實際的可用性。另外，一提到泛型，相信大家用到最多的就是在集合中，其實，在實際的程式設計過程中，自己可以使用泛型去簡化開發，且能很好的保證程式碼品質。

基本型態的包裝類別

在Number的子類別有Integer, Boolean, …