# 体验泛型

* Jdk 1.5以前的集合类中存在什么问题

ArrayList collection = new ArrayList();

collection.add(1);

collection.add(1L);

collection.add(“abc”);

int i = (Integer) collection.get(1); //编译要强制类型转换且运行时出错！

* Jdk 1.5的集合类希望你在定义集合时，明确表示你要向集合装哪种类型的数据，无法加入

指定类型以外的数据。

ArrayList<Integer> collection2 = new ArrayList<Integer>();

collection2.add(1);

/\*collection2.add(1L);

collection2.add(“abc”);\*/ //这两行代码编译时就报告了语法错误

int i2 = collection2.get(0); //不需要再进行类型转换

* 泛型是提供给javac编译器使用的，可以限定集合中的输入类型，让编译器挡住源程序中的非法输入，编译器编译带参数类型说明的集合时会去除掉“类型”信息，使程序运行效率不受影响，对于参数化的泛型类型，getClass()方法的返回值和原始类型完全一样。由于编译生成的字节码会去掉泛型的类型信息，只要能跳过编译器，就可以往某个泛型集合中加入其它类型的数据，例如，用反射得到集合，再调用其add方法即可。
* 泛型解决的不只是让您少写几个类的程序代码，还在于让您定义安全的泛型类（Generic Class）.泛型提供编译时期检查，您不会因为将对象置于某个容器(Container)而失去其类型。

# 了解泛型

* ArrayList<E>类定义和ArrayList<Integer>类引用中涉及如下术语：
* 整个ArrayList<E>称为泛型类型
* ArrayList<E>中的E称为类型变量或类型参数
* 整个ArrayList<Integer>称为参数化类型
* ArrayList<Integer>中的Integer称为类型参数的实例或实际类型参数
* ArrayList<Integer>中<>念着typeof
* ArrayList称为原始类型
* 参数化类型与原始类型的兼容性
* 参数化类型可以引用一个原始类型的对象，编译报告警告，例如，

Collection<String> c = new Vector(); // 可不可以，不就是编译器一句话的事吗？

* 原始类型可以引用一个参数化类型的对象，编译报告警告，例如，

Collection c = new Vector<String>();

//原来的方法接受一个集合参数，新的类型也要能传进去

* 参数化类型不考虑类型参数的继承关系：
* Vector<String> v = new Vector<Object>();

//错误! //不写<Object>没错，写了就是明知故犯

* Vector<Object> v = new Vector<String>(); //也错误！
* 在创建数组实例时，数组的元素不能使用参数化的类型，例如，下面语句有错误：

Vector<Integer> vectorList[] = new Vector<Integer>[10];

* 思考题：下面的代码会报错误吗？ //不会报错

Vector v1 = new Vector<String>();

Vector<Object> v = v1 ;

泛型中的类型参数严格说明集合中装载的数据类型是什么和可以加入什么类型的数据，

记住：Collection<Object> 与 Collection<String> 是两个没有转换关系的参数化类型。假设 Vector<String> v = new Vector<Object>(); 可以的话，那么以后从v中取出的对象当作String用，而 v 实际指向的对象中可以加入任意的类型的对象；假设

Vector<Object> v = new Vector<String>();可以的话，那么以后可以向v中加入任意的类型对象，而v实际指向的集合中只能装String 类型的对象。

# 由C++的模板函数引入自定义泛型

* 如下函数的结构很相似，仅类型不同：
* int add(int x , int y){

return x+y ;

}

* float add(float x, float y){

return x+y ;

}

* double add(double x , double y){

return x+y ;

}

* C++用模板函数解决，只写一个通用的方法，它可以适应各种类型，示意代码如下：

template<class T>

T add(T x , T y){

return (T)(x+y);

}

# 定义泛型类型

* 如果类的实例对象中的多处都要用到同一个泛型参数，即这些地方引用的泛型类型要保持同一个实际类型时，这时候就要采用泛型类型的方式进行定义，也就是类级别的泛型，语法格式如下：

public class GenericDao<T>{

private T field1;

public void save(T obj){};

public T getById(int id){}

}

* 类级别的泛型是根据引用该类名时指定的类型信息来参数化类型变量的，例如，如下两种方式都可以：
* GenericDao<String> dao = null;
* new GenericDao<String>();
* 注意：
* 在对泛型类型进行参数化时，类型参数的实例必须是引用类型，不能是基本类型。
* 当一个变量被声明为泛型时，只能被实例变量和方法调用（还有内嵌类型），而不能被静态变量和静态方法调用。因为静态成员是被所有参数化的类所共享的，所以静态成员不应该有类级别的类型参数。
* 问题：类中只有一个方法需要使用泛型，是使用类级别的泛型，还是使用方法级别的泛型？

应该使用方法级别的泛型。

# 定义泛型方法

* Java的泛型方法没有C++模板函数功能强大，java中的如下代码无法通过编译：

<T> T add( T x , T y) {

return (T)(x+y);

//return null;

}

* 交换数组中的两个元素的位置的泛型方法语法定义如下：

static <E> void swap(E[] a , int i , int j){

E t = a[i];

a[i] = a[j];

a[j] = t;

}

* 用于放置泛型的类型参数的尖括号应出现在方法的其他所有修饰符之后和方法的返回类型之前，也就是紧邻返回值之前。按照惯例，类型参数通常用单个大写字母表示。
* 除了在应用泛型时可以使用extends 限定符，在定义泛型时也可以使用extends限定符，例如Class.getAnnotation()方法的定义。并且可以用&来指定多个边界，如

< V extends Serializable & cloneable > void method(){}

* 普通方法、构造方法和静态方法中都可以使用泛型。编译器也不允许创建类型变量的数组。

也可以用类型变量表示异常，称为参数化的异常，可以用于方法的throws列表中，但不能用于catch子句中。

* 在泛型中可以同时有多个类型参数，在定义它们的尖括号中用逗号分割，例如：

public static <K,V> V getValue(key){return map.get(key); }

# 类型参数的类型推断

编译器判断泛型方法的实际类型参数的过程称为类型推断，类型推断是相对于知觉推断的，其实现方法是一种非常复杂的过程。

根据调用泛型方法时实际传递的参数类型或返回值的类型来推断，具体规则如下：

* 当某个类型变量只在整个参数列表中的所有参数和返回值中的一处被应用了，那么根 据调用方法时该处的实际应用类型来确定，这很容易凭着感觉推断出来，即直接根据调用 方法时传递的参数类型或返回值来决定泛型参数的类型，例如：

swap( new String[3], 3, 4 ) ---🡪 static <E> void swap(E[] a , int i , int j)

* 当某个类型变量在整个参数列表中的所有参数和返回值中的多处被应用了，如果调用方法时这多处的实际应用类型都对应同一种类型来确定，这很容易凭着感觉推断出来：

add(3 , 5 ) ---🡪 static <T> void add ( T a , T b)

* 当某个类型变量在整个参数列表中的所有参数和返回值中的多处被应用了，如果调用方法时这多处的实际应用类型对应到了不同的类型，且没有使用返回值，这时候取多个参数中的最大交集类型，例如，下面语句实际对应的类型就是Number了，编译没问题，只是运行时出问题：

fill(new Integer[3] , 3.5f) --🡪 static <T> void fill(T[], T v)

* 当某个类型变量在整个参数列表中的所有参数和返回值中的多处被应用了，如果调用方法时这多处的实际应用类型对应到了不同的类型，并且使用返回值，这时候优先考虑返回值的类型，例如，下面语句实际对应的类型就是Integer了，编译将报告错误，将变量x的类型改为float，对比eclipse报告的错误提示，接着再将变量x类型改为Number，则没有了错误：

int x = add(3 , 3.5f) --🡪 static <T> T add(T a, T b)

* 参数类型的类型推断具有传递性，下面第一种情况推断实际参数类型为Object,编译没有问题，而第二种情况则根据参数化的Vector类实例将类型变量直接确定为String类型，编译将出现问题：

copy( new Integer[5] , new String[5]) --🡪static <T> void copy(T[] a , T[] b)

copy( new Vector<String>(), new Integer[5])-🡪static <T> void copy(Collection<T> a, T[] b)

# 泛型中的?通配符

* 问题：
* 定义一个方法，该方法用于打印出任意参数化类型的集合中的所有数据，该方法如何定义呢？
* 错误方式：

public static void printCollection(Collection<Object> cols){

for(Object obj : cols){

System.out.println(obj);

}

/\*

cols.add("string"); //没错

cols = new HashSet<Date>(); //会报告错误

\*/

}

* 正确方式

public static void printCollection(Collection<?> cols){

for(Object obj : cols){

System.out.println(obj);

}

// cols.add("string"); //错误，因为它不知自己未来匹配就一定是String

cols.size(); //没错，此方法与类型参数没有关系

cols = new HashSet<Date>();

}

* 总结：
* 使用？通配符可以引用其他各种参数化的类型，？通配符定义的变量主要用作引用，可以调用与参数化无关的方法，不能调用与参数化有关的方法。

/\*Cols<Object>中的Object只是说明Cols<Object> 实例对象中的方法接受的参数是Object,

Cols<Object>是一种具体类型，new HashSet<Date>也是一种具体类型，两者没有兼容性问题。\*/

Collection<？> a 可以与任意参数化的类型匹配，但到底匹配的是什么类型，只有以后才知道，所以， a = new ArrayList<Integer>() 和 a = new ArrayList<String>()都可以，但a.add( new Date())或a.add(“abc”)都不行。

# 泛型中的?通配符的扩展

* 限定通配符的上边界：
* 正确：Vector<? extends Number> x = new Vector<Integer>();
* 错误：Vector<? extends Number> x = new Vector<String>();
* 限定通配符的下边界：
* 正确：Vector<? super Integer> x = new Vector<Number>();
* 错误：Vector<? super Integer> x = new Vector<Byte>();
* 限定通配符总是包括自己。

代码示例：

**public** **class** GenericTest {

**public** **static** **void** main(String[] args) **throws** Exception {

// **TODO** Auto-generated method stub

ArrayList collection1 = **new** ArrayList();

collection1.add(1);

collection1.add(1L);

collection1.add("abc");

// int i = (Integer)collection1.get(1);

// 泛型，不用再进行强制转换

ArrayList<String> collection2 = **new** ArrayList<String>();

// collection2.add(1);

// collection2.add(1L);

collection2.add("abc");

String element = collection2.get(0);

Constructor<String> constructor1 = String.**class**

.getConstructor(StringBuffer.**class**); // 选择String的哪一个构造方法

String str2 = constructor1.newInstance(/\*\* "abc" \*/

**new** StringBuffer("abc")); // 用这个构造方法时需要传一个StringBuffer的对象进去

System.*out*.println(str2.charAt(2));

// collection2，collection3中的泛型被javac用完之后就会去掉类型，所以两者比较其getClass()方法时

// 返回true

ArrayList<Integer> collection3 = **new** ArrayList<Integer>();

System.*out*.println(collection3.getClass() == collection2.getClass());

// collection3.add("abc");

// 泛型是给编译器javac看的，此处通过反射跳过编译器所以可以将"abc"添加进ArrayList<Integer>中

collection3.getClass().getMethod("add", Object.**class**)

.invoke(collection3, "abc");

System.*out*.println(collection3.get(0));

*printCollection*(collection3);

// Class<? extends Number> x = String.class.asSubclass(Integer.class);

Class<?> y;

Class<String> x; // Class.forName("java.lang.String");

HashMap<String, Integer> maps = **new** HashMap<String, Integer>();

maps.put("zxx", 28);

maps.put("lhm", 35);

maps.put("flx", 33);

Set<Map.Entry<String, Integer>> entrySet = maps.entrySet();

**for** (Map.Entry<String, Integer> entry : entrySet) {

System.*out*.println(entry.getKey() + ":" + entry.getValue());

}

*add*(3, 5); //自动装箱

Number x1 = *add*(3.5, 3); // 类型推断，最大公约数

Object x2 = *add*(3, "abc");

*swap*(**new** String[] { "abc", "xyz", "itcast" }, 1, 2);

//swap(new int[] { 1, 4, 3, 5, 6 }, 3, 4);

//int[]已经是引用类型，所以该数组中的元素无法再自动装箱所以编译时报错

//将Object转换成定义的类型

Object obj ="abc";

String x3 = *autoConvert*(obj);

*copy1*(**new** Vector<String>(),**new** String[10]);

*copy2*(**new** Date[10],**new** String[10]);

// copy1(new Vector<Date>(),new String[10]); 类型推断具有传播性

GenericDao<ReflectPoint> dao = **new** GenericDao<ReflectPoint>();

dao.add(**new** ReflectPoint(3,3));

// String s = dao.findById(1);

//Vector<Date> v1 = new Vector<Date>();

//如何通过反射方式得到Vector中到底存的什么东西？

//v1.getClass(); //编译后去泛型会去类型化，该方式行不通

Method applyMethod = GenericTest.**class**.getMethod("applyVector", Vector.**class**);

Type[] types = applyMethod.getGenericParameterTypes();

ParameterizedType ptype = (ParameterizedType)types[0];

System.*out*.println(ptype.getRawType()); //class java.util.Vector

System.*out*.println(ptype.getActualTypeArguments()[0]); //class java.util.Date

}

**public** **static** **void** applyVector(Vector<Date> v1){

}

**private** **static** <T> **void** fillArray(T[] a,T obj){

**for**(**int** i=0;i<a.length;i++){

a[i]=obj;

}

}

**private** **static** <T> T autoConvert(Object obj){

**return** (T)obj;

}

// 交换数组元素值，自定义泛型中的T必须是引用类型

**private** **static** <T> **void** swap(T[] a, **int** i, **int** j) {

T tmp = a[i];

a[i] = a[j];

a[j] = tmp;

}

// 泛型的定义，以及类型的推断,在返回值前面加<T>表示任意类型，接着就可以在方法体中使用

**private** **static** <T> T add(T x, T y) {

**return** **null**;

}

//用下面的代码说明对异常如何采用类型:

**private** **static** <T **extends** Exception> **void** sayHello() **throws** T{

**try**{

}**catch**(Exception e){

**throw** (T)e;

}

}

// ?通配符表示可以匹配任意类型

**public** **static** **void** printCollection(Collection<?> collection) {

// collection.add(1);

System.*out*.println(collection.size());

**for** (Object obj : collection) {

System.*out*.println(obj);

}

}

// 采用自定泛型的方式打印出任意参数化类型的集合中的所有内容

**public** **static** <T> **void** printCollection2(Collection<T> collection,T obj2) {

// collection.add(1);

System.*out*.println(collection.size());

**for** (Object obj : collection) {

System.*out*.println(obj);

}

collection.add(obj2);

}

//循环将src中内容复制到dest中

**public** **static** <T> **void** copy1(Collection<T> dest,T[] src){ }

**public** **static** <T> **void** copy2(T[] dest,T[] src){}

}

//dao -->data access object(数据访问对象)--->crud(create、read、update、delete)

**public** **class** GenericDao<E> {

**public** **void** add(E x){

}

**public** E findById(**int** id){

**return** **null**;

}

**public** **void** delete(E obj){

}

**public** **void** delete(**int** id){

}

**public** **void** update(E x){

}

//静态方法不能使用泛型类型

//public static void update2(E x){}

**public** **static** <E> **void** update2(E x){}//该方法中的泛型跟类定义的E没关系

**public** Set<E> findByConditions(String where){

**return** **null**;

}

**public** E findByUserName(String name){

**return** **null**;

}

}