**泛型**

**概念**

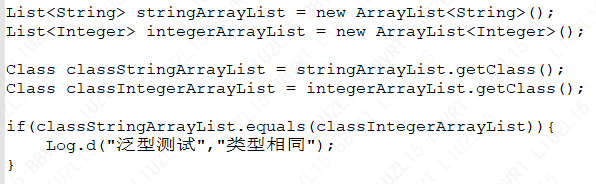
泛型，即“参数化类型”。

**目的**

主要目的之一就是用来指定容器要持有什么类型的对象，而且由编译器来保证类型的正确性。

**泛型擦除**

泛型只在编译阶段有效。在编译之后会采取去泛型化的措施。



输出结果

**泛型测试：类型相同**

在运行期我们是获取不到任何已经声明的类型信息的。

xxxList.getClass().getTypeParameters(); //返回的都是“形参”

**注意：**

编译器虽然会在编译过程中移除参数的类型信息，但是会保证类或方法内部参数类型的一致性。

**泛型参数将会被擦除到它的第一个边界**（边界可以有多个，重用 extends 关键字，通过它能给与参数类型添加一个边界）。编译器事实上会把类型参数替换为它的第一个边界的类型。如果没有指明边界，那么类型参数将被擦除到Object。下面的例子中，可以**把泛型参数T当作HasF类型**来使用。

public interface HasF {

void f();

}

public class Manipulator<T extends HasF> {

T obj;

public T getObj() {

return obj;

}

public void setObj(T obj) {

this.obj = obj;

}

}

extend关键字后后面的类型信息决定了泛型参数能保留的信息。Java类型擦除只会擦除到HasF类型。

**原理**

在编译过程中，类型变量的信息是能拿到的。所以，set方法在编译器可以做类型检查，非法类型不能通过编译。但是对于get方法，由于擦除机制，运行时的实际引用类型为Object类型。为了“还原”返回结果的类型，编译器在get之后添加了**类型转换**。所以，在GenericHolder.class文件main方法主体第18行有一处类型转换的逻辑。它是编译器自动帮我们加进去的。所以在泛型类对象读取和写入的位置为我们做了处理，为代码添加约束。

**注意事项**

泛型的类型参数智能是类类型（包括自定义类），不能是简单类型(int 等)

基本形式

<T>

每个用例的标准类型参数：

E：元素

K：键

N：数字

T：类型

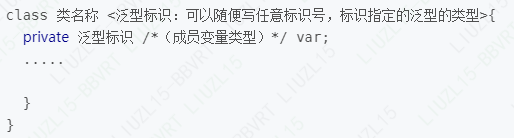
V：值

S、U、V 等：多参数情况中的第 2、3、4 个类型

**使用范围**

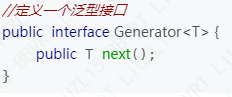
**泛型类**

基本写法



**在实例化泛型类时，必须指定T的具体类型**。

**泛型接口**



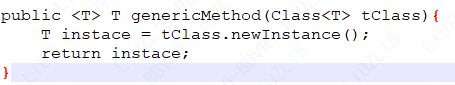
使用：

在声明类的时候，需将泛型的声明也一起加到类中。

class XXXGenerator<T> implements Generator<T>{

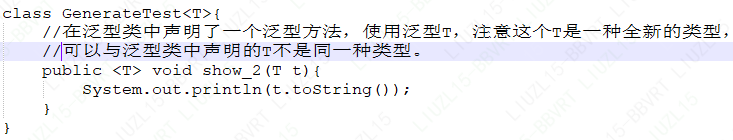
}

**泛型方法**



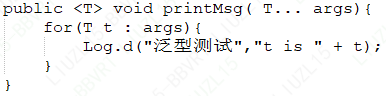
说明

1. 只有声明了<T>的方法才是泛型方法，泛型类中的使用了泛型的成员方法并不是泛型方法。
2. 同时声明了一个泛型 T
3. 及时在泛型类中声明了同样标识的泛型方法，泛型方法中的标识T也是一个全新的类型，可以与泛型类中的标识不一样的类型

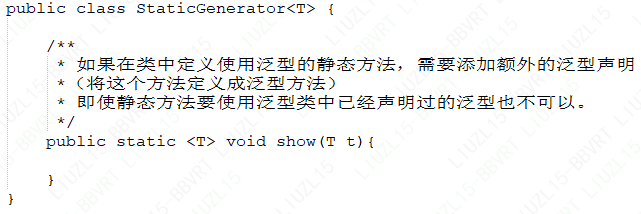


1. 泛型方法能使方法独立于类而产生变化。

泛型方法与可变参数



静态方法与泛型



**如果静态方法要使用泛型的话，必须将静态方法也定义成泛型方法**。

**泛型通配符**

类型通配符一般使用 ? 代替具体的类型实参，此处 ? 是类型实参，而不是类型形参。可以把 ? 看成所有类型的父类，是一种真实的类型。

public void showKeyValue(Generator<?> obj){ }

**泛型上下边界**

使用泛型的时候，我们还可以为传入的泛型类型实参进行上下边界的限制，如：类型实参只准传入某种类型的父类或某种类型的子类。

**为泛型添加上边界**，即传入的类型实参必须是指定类型的子类型，如：

Generic<? extends Number>

类

public class Generic<? extends Number>{}

泛型方法

public <T extends Number> T showKeyName(Generic<T> obj)

泛型形参

public void showKeyValue(Generic<? extends Number>)

注意

**设置 ? 为 Fruit 的父类**，则<? extends Fruit>会使往盘子里放东西的set()方法失效。但取东西get()方法还有效。

**为泛型添加下边界**

<? super T>，容器中只能存放T及其T的基类类型的数据。

**注意**

下界通配符<? super T>不影响往里面存储**T及其派生类**，但是读取出来的数据只能是Object类型。

**PECS原则**

上界<? extends T>不能往里存，只能往外取，适合频繁往外面读取内容的场景。

下界<? super T>不影响往里存，但往外取只能放在Object对象里，适合经常往里面插入数据的场景。

**泛型数组(容器)**

不能创建一个确切的泛型类型的数组。

这个例子是不可以的。

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

使用通配符创建泛型数组是可以的，如：

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

这样也是可以的，如：

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

示例

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

**使用场景**

**元组的使用**

泛型类最常用的使用场景就是“元组”的使用。我们知道方法return返回值只能返回单个对象。如果我们定义一个泛型类，定义2个甚至3个类型参数，这样我们return对象的时候，构建这样一个“元组”数据，通过泛型传入多个对象，这样我们就可以一次性方法多个数据了。

**泛型的局限**

**泛型类型不能显式地运用在运行时类型的操作当中**，如：转型、instanceof 和 new。下面的代码是无法通过编译的。

public class Erased<T> {

private final int SIZE = 100;

public static void f(Object arg) {

//编译不通过

if (arg instanceof T) {

}

//编译不通过

T var = new T();

//编译不通过

T[] array = new T[SIZE];

//编译不通过

T[] array = (T) new Object[SIZE];

}

}

**解决方法**

**类型判断问题**

/\*\*

\* 泛型类型判断封装类

\* @param <T>

\*/

class GenericType<T>{

Class<?> classType;

public GenericType(Class<?> type) {

classType=type;

}

public boolean isInstance(Object object) {

return classType.isInstance(object);

}

}

main方法调用

GenericType<A> genericType=new GenericType<>(A.class);

System.out.println("------------");

System.out.println(genericType.isInstance(new A()));

System.out.println(genericType.isInstance(new B()));

**通过记录类型参数的Class对象，然后通过这个Class对象进行类型判断**。

**创建类型实例**

泛型代码中不能new T()的原因有两个：

一是因为擦除，不能确定类型

二是无法确定T是否包含无参构造函数

为了避免这两个问题，可以使用显示的工厂模式：

/\*\*

\* 使用工厂方法来创建实例

\*

\* @param <T>

\*/

interface Factory<T>{

T create();

}

class Creater<T>{

T instance;

public <F extends Factory<T>> T newInstance(F f) {

instance=f.create();

return instance;

}

}

class IntegerFactory implements Factory<Integer>{

@Override

public Integer create() {

Integer integer=new Integer(9);

return integer;

}

}

通过工厂模式+泛型方法来创建实例对象。调用如下：

Creater<Integer> creater = new Creater();

System.out.println(creater.newInstance(new IntegerFactory()));

**创建泛型数组**

一般不建议创建泛型数组，尽量使用ArrayList来代替泛型数组。这里给出一种创纪检数组的方法。

public class GenericArrayWithTypeToken<T> {

private T[] array;

@SuppressWarnings("unchecked")

public GenericArrayWithTypeToken(Class<T> type, int sz) {

array = (T[]) Array.newInstance(type, sz);

}

public void put(int index, T item) {

array[index] = item;

}

public T[] rep() {

return array;

}

public static void main(String[] args) {

}

}

这里使用的还是传参数类型，利用类型的 newInstance 方法创建实例的方式。