**泛型**，即“参数化类型”。

（1）泛型的本质是为了参数化类型。也就是说在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

（2）泛型只在编译阶段有效，编译阶段之后，程序会采取去泛型化的措施。也就是说，Java中的泛型，只在编译阶段有效。在编译过程中，正确检验泛型结果后，会将泛型的相关信息擦出，并且在对象进入和离开方法的边界处添加类型检查和类型转换的方法。也就是说，泛型信息不会进入到运行阶段。

eg：

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

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

Class classA = A.getClass();

Class classB = B.getClass();

if(A.equals(B)){

Log.d(“TEST\_RESULT：”,”泛型类型相同”);

}

输出结果：D/TEST\_RESULT：泛型类型相同

（3）泛型有三种使用方式，分别为：泛型类、泛型接口、泛型方法

**泛型类**：

通过泛型可以完成对一组类的操作对外开放相同的接口。最典型的就是各种容器类，如：List、Map、Set

**泛型类最基本写法：**

class 类名 <泛型标识：可以随便写任意标识号，标识指定的泛型的类型>{

private 泛型标识 var

}

eg：一个最普通的泛型类

public class Generic<T>{//此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型

private T key; //key这个成员变量的类型为T，T的类型由外部指定

public Generic(T key){ //泛型构造方法形参key的类型也为T，T的类型由外部指定

this.key = key;

}

public T getKey(){ //泛型方法getKey的返回值类型为T，T的类型由外部指定

return key;

}

}

传入的实参类型需与泛型的类型参数类型相同，并且泛型的类型参数只能是类类型（包括自定义类），不能是简单类型

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

Generic<String> genericString = new generic<String>(“key\_value”);

定义了泛型类，不一定要传入泛型类型实参。

（1）如果在使用泛型时传入泛型实参，在编译时就会根据传入的泛型实参做相应的限制，这时泛型才能够起到限制作用。

（2）如果不传入泛型类型实参，在泛型类中使用泛型的方法或成员变量定义的类型可以为任何的类型。

Generic generic = new Generic("111111");

Generic generic1 = new Generic(4444);

Generic generic2 = new Generic(55.55);

Generic generic3 = new Generic(false);

注意：不能对确切了泛型类型参数的泛型类使用instanceof操作.

**泛型接口**

泛型接口与泛型类的定义及使用基本相同。泛型接口常被用在各种类的生产器中。

eg:

//定义一个泛型接口

public interface Generator<T> {

public T next();

}

当实现泛型接口的类，未传入泛型实参时，与泛型类的定义相同，在声明类时，需要将泛型的声明也一起加到类中

class generatorClass<T> implements Generator<T>{

@Override

public T next(){

return null;

}

}

当实现泛型接口的类，传入泛型实参时，则所有使用泛型的地方都要替换成传入的实参类型

public class generatorClass implements Generator<String>{

private String str = “入阵曲”;

@Override

public String next(){

return str;

}

}

**泛型通配符**

Integer是Number的子类，Generic<Integer>与Generic<Number>实际上是相同的一种基本类型。那么在使用Generic<Number>作为形参的方法中，能否使用Generic<Integer>的实例传入呢？即能否在逻辑上将Generic<Number>和Generic<Integer>看成具有父子关系的泛型类型呢？

eg:

public void test(Generic<Number> obj){}

Generic<Number> genNumber = new Generic<Number>(1604);

Generic<Integer> genInteger = new Generic<Integer>(1321);

test(genInteger);

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

test(genNumber);

因此Generic<Integer>不能被看做Generic<Number>子类。由此可以看出：**虽然同一种泛型可以对应多个版本（参数类型不固定），但是不同版本的泛型实例是不兼容的！**

泛型通配符？可以实现以上功能。

使用？代替具体的**类型实参**，注意不是类型形参！即？是所有类型的父类，并且是一种真实的类型。

eg:

public void test(Generic<?> obj){}

当具体类型不确定时，这个通配符就是?

当不需要使用到具体类型的功能，只使用类型父类Object的功能，可以用?通配符来表示未知类型。

**泛型方法**

泛型方法，是在**调用方法的时候指明泛型的具体类型**。

注意，假若存在以下的泛型类：

public class Generic<T> {

private T key;

public Generic(T key) {

this.key = key;

}

public T getKey(){//不是泛型方法，因为它的返回值是在声明泛型类时已经声明过的泛型，所以这个方法可以直接使用T这个泛型

return key;

}

}

**泛型方法介绍：**

eg：

public <T> T genericMethod(Class<T> tClass)

throws InstantiationException,IllegalAccessException{

T instance = tClass.newInstance();

return instance;

}

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

首先public与返回值之间的<T>必不可少，表明这是一个泛型方法，并且声明了一个泛型T，这个T可以出现在该泛型方法的任意位置，泛型的数量也可以为任意多个。

public <T,K> K genericMethod(Generic<T> instance){

……

……

}

**类中的泛型方法（特殊）**

class GenerateTest<T>{

//该成员方法不是泛型方法

//该方法形参类型T与泛型类的泛型类型T一致，由泛型类决定

public void genericMethod (T t){

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

}

//在泛型类型为T的泛型类中声明一个泛型方法，使用泛型E

//由于泛型方法在声明的时候声明了泛型<E>，因此即使在泛型类中并未声明泛型，编译器也能够正确识别泛型方法中识别的泛型

public <E> void genericMethod1(E t){}

//在泛型类型为T的泛型类中声明一个泛型方法，使用泛型T

//注意这个T是一种全新的类型，由于这里是在方法上声明的，可以是与泛型类中声明的T不是同一种类型

public <T> void genericMethod2(T t){}

}

**泛型方法与可变参数**

public <T> void getMsg(T… args){

for(T t: args){

System.out.println(“”+t);

}

}

**静态方法与泛型**

静态方法有一种情况需要注意，就是类中的静态方法无法访问类上定义的类型。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。

public class StaticGenerator<T>{

…

…

public static <T> void generaticMethod(T t){}

/\*public static void wrongMethod(T t) 该方法报错\*/

}

**泛型上下边界**

在使用泛型时，还可以给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。

添加上边界：即传入的类型实参必须是指定类型的子类型

（1）

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

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

//test(generic1); 报错

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

test(generic2);

Generic<Float> generic3 = new Generic<Float>(2.4f);

test(generic3);

Generic<Double> generic4 = new Generic<Double>(2.56);

test(generic4);

（2）

public class Generic<T extends Number>{

private T key;

public Generic(T key){

this.key = key;

}

public T getKey(){

return key;

}

}

//Generic<String> generic = new Generic<String>(“hjs”); 报错

（3）

//在泛型方法中添加上下边界限制时，必须在权限声明与返回值之间的<T>上添加上下边界

public <T extends Number> T Generic(Generic<T> t){

}

总结：泛型上下边界的添加，必须与泛型的声明联系在一起。

**泛型数组**

（1）java中是**不能创建一个确切泛型类型**的数组的

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

而这种做法允许：

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

（2）而使用通配符创建泛型数组是可以的

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

map.entrySet()是将map里的每一个键值对取出来封装成一个Entry对象再存到一个Set里。

Map.Entry<String,String>表示一个泛型