**中软国际考题-笔试**

1. 单选题

1、创建一个只能存放String的泛型ArrayList的语句是哪项（B）

A. ArrayList<int> al = new ArrayList<int>();

B. ArrayList<String> al = new ArrayList<String>();

C. ArrayList al = new ArrayList<String>();

D. ArrayList<String> al = new List<String>();

2、下列描述错误的是（B）

A.泛型往往与集合同时声明创建

B.泛型定义包括泛型类定义和泛型对象定义，其中泛型类 定义的应用最为普遍

C.泛型规定了集合中的存储子元素

D.泛型自身可以叠加规定

3、下列描述正确的是（A）

1. 泛型可以提高数据安全性
2. 规定泛型后就不需要数据强转了
3. 泛型不是必须要编写的，所以可用可不用。
4. 泛型只能规定基本数据类型

4、java中泛型描述错误的是（D）

A.泛型可以规定class类

B.泛型可以规定map键值对

C.泛型对组成元素存储具有强制作用

D.泛型与基本数据类型没有直接联系

5、以下哪个不是Java中使用泛型的好处：D

A: 类型安全，提供编译期间的类型检测

B: 前后兼容

C: 泛化代码,代码可以更多的重复利用

D: 节约代码

6、泛型使用错误的方式（B）

A.泛型方法 B.泛型参数 C.泛型接口 D泛型类

7、泛型的<>中放的是什么数据类型（B）

A.引用数据类型 B.基本数据类型 C.泛型类型 D.注解类型

8、定义一个List集合对象， 集合内存放String类型单元。 （ B ）

A. List<String> list = new ArrayList<String>;

B. List<String> list = new ArrayList<>();

C. List<String> list = new ArrayList<String>;

D. List<String> list = new ArrayList(String);

9. 定义一个Map集合对象， 单元的key是整数， value是字符串。（ D ）

A. Map<Integer, String> set = new HashMap<Integer, String>;

B. Map<String, Integer> set = new HashMap<>();

C. Map<Integer, String> set = new HashMap<>(Integer, String);

D. Map<Integer, String> set = new HashMap<Integer, String>();

10、下列关于反射和泛型的联系说法正确的是( B )   
A、泛型只在编译时有效，无法在运行期获取泛型的具体类型  
B、反射机制可以获取泛型的具体类型  
C、可以通过反射绕过泛型检查，因为运行期泛型根本没有用  
D、以上说法都不对

11、下面哪一个是设定泛型的下界（B）

A. List<? extends T>

B. List<? super T>

C. List< super >

D. List< extends >

12、 下列代码的执行结果为（C）

public class Test {

public static void main(String[] args) {

GT<Integer> gti = new GT<Integer>();

gti.var = 1;

GT<String> gts = new GT<String>();

gts.var = 2;

System.out.println(gti.var);

}

}

class GT<T>{

public static int var = 0;

public void nothing(T t) {

}

}

A.1 B.0 C.2 D.编码错误

1. 多选题
2. 下列哪些项是泛型的优点？（A、C）

A. 不用向下强制类型转换 B. 代码容易编写

C. 类型安全 D. 运行速度快

2、下面哪些项描述不正确？（A、B）

A. 泛型无法规定方法的形式参数

B. 泛型只允许单一编写规定

C. 方法返回值也可以由泛型规定

D. 泛型允许编写自定义类

3、Java的泛型是如何工作的 ?（ABCD）

a)类型检查：在生成字节码之前提供类型检查

b)类型擦除：所有类型参数都用他们的限定类型替换，包括类、变量和方法（类型擦除）

c)如果类型擦除和多态性发生了冲突时，则在子类中生成桥方法解决

d)如果调用泛型方法的返回类型被擦除，则在调用该方法时插入强制类型转换

4、泛型使用的注意事项有哪些？（A、B、C、D）

A. 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

B. 泛型的类型参数可以有多个

C. 不能对确切的泛型类型使用instanceof操作。

D. 不能创建一个确切的泛型类型的数组

5、以下哪些类型的数据可以放在泛型的<>中？（A、B）

A. int B. Intger C. Class Test D. @getPost;

6、下面描述中正确的是？（A、C、D）

A. class MyClass<T>表示定义了一个泛型类

B. 类型擦除在编译阶段、运行阶段都有效。

C、interface IMessage<T>表示定义了一个泛型接口

D. ？ super String : 表示此方法只能取得String以及其父类Object。

7、下面代码正确的有？（A、B、D）

A. public class Matrix<T extends Numbers>{…}

B. public class Matrix<? super Number>{…}

C. public class Matrix<? implements Number>{…}

D. public void add(Matrix<?> m){...}

8、判断以下哪种书写是正确的？（A、D、E）

A. ArrayList<String> lists = new ArrayList<String>()

B. ArrayList<Object> lists = new ArrayList<String>()

C. ArrayList<String> lists = new ArrayList<Object>()

D. ArrayList<String> lists = new ArrayList()

E. ArrayList lists = new ArrayList<String>()

1. 填空题（每空1分，共10分）
2. Java泛型体系中的\_\_\_\_\_\_\_\_\_\_指的是通过类型参数合并，将泛型类型实例关联到同一份字节码上。编译器只为泛型类型生成一份字节码，并将其实例关联到这份字节码上 (类型擦除)

2、泛型语法为：类名称<\_\_\_\_\_\_\_\_\_> \_\_\_\_\_\_\_\_\_ = new \_\_\_\_\_\_\_\_\_<\_\_\_\_\_\_\_\_\_>();。(具体类，对象名称，类名称，具体类)

3、泛型是通过\_\_\_\_\_\_\_\_\_\_ 来实现的，\_\_\_\_\_\_\_\_\_\_在编译时擦除了所有类型相关的信息，所以在运行时不存在任何类型相关的信息。 （类型擦除，编译器）

4、java在接口、类或类的方法的声明中，声明一个泛型。多个参数时要用\_\_\_\_\_\_\_\_\_隔开。 (逗号)

5、泛型可以声明在那三个位置：\_\_\_\_\_\_\_、\_\_\_\_\_\_\_\_\_\_ 、\_\_\_\_\_\_\_\_\_。(类、接口、方法)

6、泛型编程中 ？代表 。不确定的java类型;

7、泛型编程中 T(type) 代表 。 具体的一个java类型

8、泛型编程中 K V (key value) 代表 。 java键值中的Key类型和 Value 类型

9、 ? extends E 表示 。 接收E类型或者E的子类型

10、 ? super E 表示 。接收E类型或者E的父类型

11、泛型中的T表示\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。(**T 被称为类型参数，用于指代任何类型**)

1. 使用泛型的好处：\_\_\_\_\_\_\_\_\_\_ 、\_\_\_\_\_\_\_\_\_、提高性能。(类型安全、消除强制类型转换)

13、所有泛型类的类型参数在编译时都会被\_\_\_\_\_\_\_\_\_\_ 。(擦除)

1. 判断题
2. 可以把List<String>传递给一个接受List<Object>参数的方法。（ F ）
3. 泛型无法在类的实例化中编写。（ N ）
4. 调用泛型方法的返回类型被擦除，则在调用该方法时插入强制类型转换。（ T ）
5. Array中可以用泛型。（ F ）
6. 可以把List<String>传递给一个接受List<Object>参数的方法（ F ）
7. 继承泛型类，泛型接口时，可以指明泛型的类型（ T ）
8. 静态方法中可以使用泛型（F）
9. catch中不能使用泛型（T）
10. 使用通配符的泛型集合可以遍历，但是不能添加（F）
11. 泛型 Generic type是JDK1.5引入的一个新特性。（ T ）
12. ? extends Number : 表示泛型必须是Number子类。（T）
13. Array中可以用泛型吗。（ F ）
14. 可以把List<String>传递给一个接受List<Object>参数的方法。（ F ）
15. Array中不可以使用泛型。（ T ）

五、简答题

1、Java中的泛型是什么 ? 使用泛型的好处是什么?

答案：泛型是Java SE 1.5的新特性，泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。它带来的好处有：

1、类型安全，提供编译期间的类型检测

2、前后兼容

3、泛化代码,代码可以更多的重复利用

4、性能较高，用GJ(泛型JAVA)编写的代码可以为java编译器和虚拟机带来更多的类型信息，这些信息对java程序做进一步优化提供条件。

2、Java的泛型是如何工作的 ? 什么是类型擦除 ?

答案：泛型如何工作：

1、类型检查：在生成字节码之前提供类型检查

2、类型擦除：所有类型参数都用他们的限定类型替换，包括类、变量和方法（类型擦除）

3、如果类型擦除和多态性发生了冲突时，则在子类中生成桥方法解决

4、如果调用泛型方法的返回类型被擦除，则在调用该方法时插入强制类型转换

类型擦除：

所有类型参数都用他们的限定类型替换，比如T->Object ? extends BaseClass->BaseClass

泛型是通过类型擦除来实现的，编译器在编译时擦除了所有类型相关的信息，所以在运行时不存在任何类型相关的信息。例如 List<String>在运行时仅用一个List来表示。这样做的目的，是确保能和Java 5之前的版本开发二进制类库进行兼容。你无法在运行时访问到类型参数，因为编译器已经把泛型类型转换成了原始类型。

3、什么是泛型中的限定通配符和非限定通配符 ?

答案：有两种限定通配符，一种是<? extends T>它通过确保类型必须是T的子类来设定类型的上界，另一种是<? super T>它通过确保类型必须是T的父类来设定类型的下界。泛型类型必须用限定内的类型来进行初始化，否则会导致编译错误。另一方面<?>表 示了非限定通配符，因为<?>可以用任意类型来替代

4、Java中List<?>和List<Object>之间的区别是什么?

答案：List<?> 是一个未知类型的List，而List<Object> 其实是任意类型的List。你可以把List<String>, List<Integer>赋值给List<?>，却不能把List<String>赋值给 List<Object>。

5、请简述泛型运用原理（5分）

答案：1.泛型的主要原理是声明类时通过标识表示类中某个属性的类型，或某方法的返回值及参数类型。此时声明和实例化类时只要指定需要的类型即可

1. 泛型定义包括泛型类定义和泛型对象定义，其中泛型对象 定义的应用最为普遍

6、如何阻止Java中的类型未检查的警告?（5分）

答案：

如果你把泛型和原始类型混合起来使用，例如下列代码，java 5的javac编译器会产生类型未检查的警告，例如

List<String> rawList = newArrayList()

注意: Hello.java使用了未检查或称为不安全的操作;

这种警告可以使用@SuppressWarnings(“unchecked”)注解来屏蔽。

7、Java中List<Object>和原始类型List之间的区别?（5分）

答案：

原始类型和带参数类型<Object>之间的主要区别是，在编译时编译器不会对原始类型进行类型安全检查，却会对带参数的类型进行检查，通过使用Object作为类型，可以告知编译器该方法可以接受任何类型的对象，比如String或Integer。

        这道题的考察点在于对泛型中原始类型的正确理解。它们之间的第二点区别是，你可以把任何带参数的类型传递给原始类型List，但却不能把List<String>传递给接受 List<Object>的方法，因为会产生编译错误。

8、什么是泛型中的限定通配符和非限定通配符 ?

答案：限定通配符对类型进行了限制。有两种限定通配符，一种是<? extends T>它通过确保类型必须是T的子类来设定类型的上界，另一种是<? super T>它通过确保类型必须是T的父类来设定类型的下界。泛型类型必须用限定内的类型来进行初始化，否则会导致编译错误。另一方面<?>表 示了非限定通配符，因为<?>可以用任意类型来替代。

9、List<? extends T>和List <? super T>之间有什么区别?

答案：这两个List的声明都是 限定通配符的例子，List<? extends T>可以接受任何继承自T的类型的List，而List<? super T>可以接受任何T的父类构成的List。例如List<? extends Number>可以接受List<Integer>或List<Float>。

10、 Java的泛型是如何工作的 ? 什么是类型擦除 ？

答案：泛型是通过类型擦除来实现的，编译器在编译时擦除了所有类型相关的信息，所以在运行时不存在任何类型相关的信息。例如 List<String>在运行时仅用一个List来表示。这样做的目的，是确保能和Java 5之前的版本开发二进制类库进行兼容。你无法在运行时访问到类型参数，因为编译器已经把泛型类型转换成了原始类型。

11、 Java中的泛型是什么 ? 使用泛型的好处是什么？

答案：1. 解决元素存储的安全性问题

2. 解决获取数据元素时，需要类型强转的问题

泛型，JDK1.5新加入的，解决数据类型的安全性问题，其主要原理是在类声明时通过一个标识表示类中某个属性的类型或者是某个方法的返回值及参数类型。这样在类声明或实例化

时只要指定好需要的具体的类型即可。

Java泛型可以保证如果程序在编译时没有发出警告，运行时就不会产生类型异常。同时，代码更加简洁、健壮。

12、 什么是泛型中的限定通配符和非限定通配符 ?

答案：限定通配符对类型进行了限制。有两种限定通配符，一种是<? extends T>它通过确保类型必须是T的子类来设定类型的上界，另一种是<? super T>它通过确保类型必须是T的父类来设定类型的下界。泛型类型必须用限定内的类型来进行初始化，否则会导致编译错误。另一方面<?>表 示了非限定通配符，因为<?>可以用任意类型来替代

13、 可以把List<String>传递给一个接受List<Object>参数的方法吗？

答案：乍看起来String是一种Object，所以 List<String>应当可以用在需要List<Object>的地方，但是事实并非如此。真这样做的话会导致编译错误。深一步考虑，就会发现Java这样做是有意义的，因为List<Object>可以存储任何类型的对象包括String, Integer等等，而List<String>却只能用来存储Strings。

14、Java为什么要引入泛型编程机制？

答：

泛型编程机制是在 JDK 5.0中引入的。 引入的目的在于减少代码错误， 提高代码质量， 并且在类型之上添加一个额外的抽象层。

15、 什么是类型擦除， 它的作用是什么？

答：在Java中， 泛型确保类型安全， 并且在运行时不会产生额外负担。编译器在编译泛型时， 使用一个称为类型擦除的步骤。

类型擦除确保删除所有的类型参数， 并使用类型参数约束或Object替代它。因此在编译后，字节码中仅包含类， 接口， 方法， 确保没有新的类型产生。在编译时应用合适的转换。

比如：

public <T> List<T> genericMethod(T t) {

return list.stream().collect(Collectors.toList());

}

在编译时， 使用Object代替无约束类型T：

public List<Object> fromArrayToList(Object a) {

return list.stream().collect(Collectors.toList());

}

如果类型T是约束的， 将使用约束替代：

public <T extends Building> void genericMethod(T t) {

...

}

编译后， 为：

public void genericMethod(Building t) {

...

}

16， 谈谈 List<T>，List<Object>，List<?>区别。

答：

ArrayList<T> al=new ArrayList<T>(); 指定集合元素只能是T类型

ArrayList<?> al=new ArrayList<?>(); 集合元素可以是任意类型

Object和T不同点在于，Object是一个实打实的类,并没有泛指谁，而T是泛指的

?和T区别是？是一个不确定类，？和T都表示不确定的类型 ，但如果是T的话，函数里面可以对T进行操作，比方 T car = getCar()，而不能用？ car = getCar()。

17， Class<T>和Class<?>有什么区别呢？它们一般应用在哪些场合？

答：T是一种具体的类，例如String， Integer等。

Class是一个类，Class对象里面存放了某个具体类的信息。

获取Class对象的三种方法：

List list = null;

Class clazz = list.getClass();

Class clazz = Class.forName("com.example.Hello");

Class clazz = List.class;

使用Class<T>和Class<?>多发生在反射场景。

Class<T>的使用：

public class Test {

public static <T> T createInstance(Class<T> clazz) throws IllegalAccessException, InstantiationException {

return clazz.newInstance();

}

public static void main(String[] args) throws IllegalAccessException, InstantiationException {

Fruit fruit= createInstance(Fruit .class);

Hello hello= createInstance(Hello.class);

}

}

Class<T>和Class<?>的区别：

Class<T>在实例化的时候，T要替换成具体类

Class<?>它是个通配泛型，?可以代表任何类型，主要用于声明时的限制情况

例如可以声明一个

public Class<?> clazz;

但是你不能声明一个

public Class<T> clazz;

18、谈谈你所理解的类型擦除。

答案：在生成的Java字节代码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会被编译器在编译的时候去掉。这个过程就称为类型擦除。如在代码中定义的List<Object>和List<String>等类型，在编译之后都会变成List。JVM看到的只是List，而由泛型附加的类型信息对JVM来说是不可见的。Java编译器会在编译时尽可能的发现可能出错的地方，但是仍然无法避免在运行时刻出现类型转换异常的情况。类型擦除也是Java的泛型实现方式与[C++模板机制](http://www.cplusplus.com/doc/tutorial/templates/)实现方式之间的重要区别。

19、简述如何定义一个泛型方法。

答案：

public <T> void Mythod(T t) {

System.out.print(t);

}

当泛型类与泛型方法共存时，泛型类中的泛型参数与泛型方法中的类型参数没有关系，泛型方法始终以自己定义的类型参数为准。

20、 List<? extends T>和List <? super T>之间有什么区别 ?

答案：List<? extends T>可以接受任何继承自T的类型的List，而List<? super T>可以接受任何T的父类构成的List。例如List<? extends Number>可以接受List<Integer>或List<Float>。

21、Java的泛型是如何工作的 ? 什么是类型擦除 ?

泛型是通过类型擦除来实现的，编译器在编译时擦除了所有类型相关的信息，所以在运行时不存在任何类型相关的信息。例如List<String>在运行时仅用一个List来表示。这样做的目的，是确保能和Java 5之前的版本开发二进制类库进行兼容。你无法在运行时访问到类型参数，因为编译器已经把泛型类型转换成了原始类型。

22、Java中List<Object>和原始类型List之间的区别?

原始类型和带参数类型<Object>之间的主要区别是，在编译时编译器不会对原始类型进行类型安全检查，却会对带参数的类型进行检查，通过使用Object作为类型，可以告知编译器该方法可以接受任何类型的对象，比如String或Integer。这道题的考察点在于对泛型中原始类型的正确理解。它们之间的第二点区别是，你可以把任何带参数的类型传递给原始类型List，但却不能把List<String>传递给接受List<Object>的方法，因为会产生编译错误。

23、List<? extends T>和List <? super T>之间有什么区别?（5分）

答案：List<? extends T>可以接受任何继承自T的类型的List, 而List<? super T>可以接受任何T的父类构成的List。

24、Java中的泛型是什么 ? 使用泛型的好处是什么?

答案：泛型是Java SE 1.5的新特性，泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。

好处：

1、类型安全，提供编译期间的类型检测

2、前后兼容

3、泛化代码,代码可以更多的重复利用

4、性能较高，用泛型编写的代码可以为java编译器和虚拟机带来更多的类型信息，这些信息对java程序做进一步优化提供条件。

25、什么是泛型中的限定通配符和非限定通配符 ?

答案：有两种限定通配符，一种是<? extends T>它通过确保类型必须是T的子类来设定类型的上界，另一种是<? super T>它通过确保类型必须是T的父类来设定类型的下界。泛型类型必须用限定内的类型来进行初始化，否则会导致编译错误。另一方面<?>表 示了非限定通配符，因为<?>可以用任意类型来替代。

26、什么是类型擦除？

答案：类型擦除指的是通过类型参数合并，将泛型类型实例关联到同一份字节码上。编译器只为泛型类型生成一份字节码，并将其实例关联到这份字节码上。类型擦除的关键在于从泛型类型中清除类型参数的相关信息，并且在必要的时候添加类型检查和类型转换的方法

六、程序实现题

1、自行实现一个带有泛型的集合类，要求实现List接口的主要方法。

|  |
| --- |
| **public** **class** MyList<T> **implements** List<T>, RandomAccess, Cloneable,  java.io.Serializable {  /\*\*  \*  \*/  **private** **static** **final** **long** *serialVersionUID* = 1L;  **private** T[] objs; // 数组容器对象  **private** **final** **int** BUF = 16; // 数组初始长度，默认为16  **private** **final** **double** EXT = 2; // 如果不够,默认增加的长度为原数组的2倍  **private** **final** **double** PER = 0.5; // 空白处大于这个百分比就缩短长度  **private** **int** count = 0; // 实际的长度(去掉尾部NULL元素)  /\*\*  \* 建立一个初始长度为10的容器  \*/  @SuppressWarnings("unchecked")  **public** MyList() {  objs = (T[]) **new** Object[BUF];  }  **private** MyList(T[] obj) {  objs = obj.clone();  }  /\*\*  \* 返回实际集合长度  \*/  @Override  **public** **int** size() {  **return** count;  }  /\*\*  \* 集合是否为空(长度是否为0)  \*/  @Override  **public** **boolean** isEmpty() {  **return** count == 0;  }  /\*\*  \* 如果列表包含指定的元素，则返回 true  \*/  @Override  **public** **boolean** contains(Object o) {  **for** (Object obj : objs) {  **if** (o.equals(obj))  **return** **true**;  }  **return** **false**;  }  /\*\*  \* 返回此集合的迭代器  \*/  @Override  **public** Iterator<T> iterator() {  **return** **new** Itr();  }  /\*\*  \* 返回数组  \*/  @Override  **public** T[] toArray() {  **return** newArray();  }  /\*\*  \* 返回按适当顺序（从第一个元素到最后一个元素） 包含列表中所有元素的数组；返回数组的运行时类型是指定数组的运行时类型。  \*/  @SuppressWarnings({ "hiding", "unchecked" })  @Override  **public** <T> T[] toArray(T[] a) {  **if** (a.length <= count) {  **return** (T[]) newArray();  } **else** {  T[] obj = (T[]) newArray();  **for** (**int** i = 0; i < a.length; i++) {  **if** (i <= obj.length)  a[i] = obj[i];  **else**  a[i] = **null**;  }  **return** obj;  }  }  /\*\*  \* 添加元素到尾部  \*/  @Override  **public** **boolean** add(T e) {  **if** (e == **null**)  **throw** **new** NullPointerException("指定的对象为空");  extSize();  objs[count] = e;  count++;  **return** **true**;  }  /\*\*  \* 移除指定的第一个出现的元素  \*/  @Override  **public** **boolean** remove(Object o) {  **if** (o == **null**)  **throw** **new** NullPointerException("指定的对象为null");  **return** delete(o);  }  /\*\*  \* 如果列表包含指定 collection 的所有元素，则返回 true。  \*/  @SuppressWarnings({ "unchecked", "rawtypes" })  @Override  **public** **boolean** containsAll(Collection<?> c) {  T[] objsclone = objs.clone();  **for** (Iterator it = c.iterator(); it.hasNext();) {  **boolean** flag = **false**;  T o = (T) it.next();  **if** (o == **null**)  **throw** **new** NullPointerException("指定的对象有空元素");  **for** (**int** i = 0; i < objsclone.length; i++) {  **if** (o.equals(objsclone[i])) {  objsclone[i] = **null**;  flag = **true**;  **break**;  }  }  **if** (!flag)  **return** **false**;  }  **return** **true**;  }  /\*\*  \* 合并指定集合到尾部  \*/  @SuppressWarnings({ "unchecked", "rawtypes" })  @Override  **public** **boolean** addAll(Collection<? **extends** T> c) {  **if** (c.size() == 0)  **return** **false**;  changeArray(count + c.size() + BUF);  **int** x = count - 1;  **for** (Iterator it = c.iterator(); it.hasNext(); x++) {  objs[x] = (T) it.next();  count++;  }  **return** **true**;  }  /\*\*  \* 添加一组元素到指定位置  \*/  @SuppressWarnings({ "unchecked", "rawtypes" })  @Override  **public** **boolean** addAll(**int** index, Collection<? **extends** T> c) {  **if** (index < 0)  **throw** **new** IndexOutOfBoundsException("角标不能小于零");  **else** **if** (index > objs.length - 1)  **throw** **new** IndexOutOfBoundsException("角标超出范围");  move(index, c.size());  **if** (objs[index] != **null**)  **throw** **new** NullPointerException("原位置不为空");  Iterator it = c.iterator();  **for** (**int** i = index; it.hasNext(); i++) {  objs[i] = (T) it.next();  }  **return** **true**;  }  /\*\*  \* 删除含有指定列表的元素  \*/  @Override  **public** **boolean** removeAll(Collection<?> c) {  **for** (Object o : c) {  **if** (!delete(o))  **throw** **new** NullPointerException("此列表包含了一个空元素");  }  **return** **true**;  }  /\*\*  \* 仅在列表中保留指定 collection 中所包含的元素,取交集  \*/  @SuppressWarnings({ "unchecked" })  @Override  **public** **boolean** retainAll(Collection<?> c) {  **int** len = count < c.size() ? count : c.size();  T[] objsclone = (T[]) **new** Object[len];  **int** x = 0;  **for** (Iterator<T> it = (Iterator<T>) c.iterator(); it.hasNext();) {  **for** (**int** i = 0; i < count; i++) {  Object o = it.next();  **if** (o == **null**)  **continue**;  **if** (o.equals(objs[i])) {  objsclone[x] = objs[i];  objs[i] = **null**;  x++;  }  }  }  objs = objsclone;  count = x;  **return** **true**;  }  /\*\*  \* 清空集合  \*/  @SuppressWarnings("unchecked")  @Override  **public** **void** clear() {  **if** (objs.length == BUF && count == 0)  **return**;  objs = (T[]) **new** Object[BUF];  count = 0;  }  /\*\*  \* 获取指定下标的元素  \*/  @Override  **public** T get(**int** index) {  **if** (index < 0)  **throw** **new** IndexOutOfBoundsException("下标不能小于零");  **else** **if** (index > objs.length - 1)  **throw** **new** IndexOutOfBoundsException("下标超出范围");  **return** objs[index];  }  /\*\*  \* 设置指定下标的值,返回以前的的值  \*/  @Override  **public** T set(**int** index, T element) {  **if** (element == **null**)  **throw** **new** NullPointerException("指定的对象为空");  **else** **if** (index < 0)  **throw** **new** IndexOutOfBoundsException("下标不能小于零");  **else** **if** (index > objs.length - 1)  **throw** **new** IndexOutOfBoundsException("下标超出范围");  T old = objs[index];  objs[index] = element;  **return** old;  }  /\*\*  \* 添加元素到指定位置  \*/  @Override  **public** **void** add(**int** index, T element) {  **if** (element == **null**)  **throw** **new** NullPointerException("指定的对象为空");  move(index, 1);  **if** (objs[index] != **null**)  **throw** **new** NullPointerException("原位置不为空");  objs[index] = element;  count++;  }  /\*\*  \* 移除指定下标元素,返回移除的值  \*/  @Override  **public** T remove(**int** index) {  **if** (index < 0)  **throw** **new** IndexOutOfBoundsException("角标不能小于零");  **else** **if** (index > objs.length - 1)  **throw** **new** IndexOutOfBoundsException("角标超出范围");  T old = delete(index);  **return** old;  }  /\*\*  \* 返回此列表中第一次出现的指定元素的索引；如果此列表不包含该元素，则返回 -1。  \*/  @Override  **public** **int** indexOf(Object o) {  **if** (o == **null**)  **throw** **new** NullPointerException("指定的对象为空");  **for** (**int** i = 0; i < objs.length; i++) {  **if** (objs[i].equals(o))  **return** i;  }  **return** -1;  }  /\*\*  \* 返回此列表中最后出现的指定元素的索引；如果列表不包含此元素，则返回 -1。  \*/  @Override  **public** **int** lastIndexOf(Object o) {  **if** (o == **null**)  **throw** **new** NullPointerException("指定的对象为空");  **for** (**int** i = objs.length - 1; i <= 0; i--) {  **if** (objs[i].equals(o))  **return** i;  }  **return** -1;  }  /\*\*  \* 返回此集合的ListIterator迭代器  \*/  @Override  **public** ListIterator<T> listIterator() {  **return** **new** listItr();  }  /\*\*  \* 返回一个ListIterator迭代器,初始位置在index  \*/  @Override  **public** ListIterator<T> listIterator(**int** index) {  **return** **new** listItr(index);  }  /\*\*  \* 返回列表中指定的 fromIndex（包括 ）和 toIndex（不包括）之间的部分视图。  \*/  @SuppressWarnings({ "unchecked", "rawtypes" })  @Override  **public** List<T> subList(**int** fromIndex, **int** toIndex) {  **if** (fromIndex > toIndex)  **return** **null**;  **else** **if** (count == 0)  **return** **null**;  **else** **if** (fromIndex < 0)  **return** **null**;  T[] arr = (T[]) **new** Object[toIndex - fromIndex + 1];  **int** x = 0;  **for** (**int** i = 0; i < count; i++) {  **if** (i >= fromIndex && i < toIndex) {  arr[x++] = objs[i];  }  }  **return** **new** MyList(arr);  }  /\*\*  \* 返回指定格式文本 [element1,element2,...]  \*/  @Override  **public** String toString() {  String str = "[";  **for** (**int** i = 0; i < count; i++) {  **if** (i == count - 1)  str += objs[i];  **else**  str += objs[i] + ",";  }  str += "]";  **return** str;  }  // Iterator迭代器  **private** **class** Itr **implements** Iterator<T> {  **private** **int** cor = -1; // 指针位置  // 是否有下一个  @Override  **public** **boolean** hasNext() {  **int** index = cor + 1;  **if** (index < 0 || index >= count)  **return** **false**;  **return** **true**;  }  // 下一个  @Override  **public** T next() {  **if** (!hasNext())  **return** **null**;  **return** objs[++cor];  }  // 删除  @Override  **public** **void** remove() {  delete(cor);  }  }  // ListIterator迭代器  **private** **class** listItr **implements** ListIterator<T> {  **private** **int** cor; // 指针位置  listItr() {  cor = -1;  }  listItr(**int** index) {  cor = index - 1;  }  // 是否有下个元素  @Override  **public** **boolean** hasNext() {  **int** index = cor + 1;  **if** (index < 0 || index >= count)  **return** **false**;  **return** **true**;  }  // 返回列表中的下一个元素。  @Override  **public** T next() {  **if** (!hasNext())  **return** **null**;  **return** objs[++cor];  }  // 前面是否有元素  @Override  **public** **boolean** hasPrevious() {  **int** index = cor - 1;  **if** (index < 0 || index >= count)  **return** **false**;  **return** **true**;  }  // 返回列表中的前一个元素。  @Override  **public** T previous() {  **if** (!hasPrevious())  **return** **null**;  **return** objs[--cor];  }  // 获取前面一个元素的位置  @Override  **public** **int** nextIndex() {  **return** cor + 1;  }  // 获取后面一个元素的位置  @Override  **public** **int** previousIndex() {  **return** cor;  }  // 删除迭代处的元素  @Override  **public** **void** remove() {  delete(cor);  }  // 设置迭代处位置新的值  @Override  **public** **void** set(T e) {  **if** (e == **null**)  **throw** **new** NullPointerException("指定的对象为空");  objs[cor] = e;  }  // 添加到尾部  @Override  **public** **void** add(T e) {  **if** (e == **null**)  **throw** **new** NullPointerException("指定的对象为空");  extSize();  objs[count] = e;  count++;  }  }  **private** T delete(**int** index) {  // 删除指定位置的元素,后面的往前移动  T old = objs[index];  **if** (index == count - 1) {  objs[index] = **null**;  count--;// 实际长度减少1  removeNull();  **return** old;  }  **for** (; index < count - 1; index++) {  objs[index] = objs[index + 1];  }  objs[count - 1] = **null**;  count--;// 实际长度减少1  removeNull();  **return** old;  }  **private** **boolean** delete(Object o) {  // 删除指定的找到的第一个元素,后面的往前移动  **if** (o == **null**)  **return** **false**;  **for** (**int** i = 0; i < objs.length - 1; i++) {  **if** (o.equals(objs[i])) {  **return** delete(i) != **null**;  }  }  **return** **false**;  }  **private** **void** move(**int** index, **int** length) {  // 从指定位置往后移动  changeArray(objs.length + length);  **for** (**int** i = count - 1; i >= index; i--) {  objs[i + length] = objs[i];  objs[i] = **null**;  }  }  **private** **void** extSize() {  // 数组长度扩充  **if** (objs.length == count)  changeArray((**int**) (count \* EXT) + BUF + count);  }  @SuppressWarnings("unchecked")  **private** **void** changeArray(**int** length) {  // 改变数组,(加长或缩短)  T[] ar = (T[]) **new** Object[length];  **for** (**int** i = 0; i < count; i++) {  ar[i] = objs[i];  }  objs = ar;  }  **private** **void** removeNull() {  // 缩小长度  **if** ((**double**) count / objs.length < PER)  changeArray(count + BUF);  }  @SuppressWarnings("unchecked")  **private** T[] newArray() {  // /复制一个数组,返回有效长度  **if** (objs.length == count)  **return** objs.clone();  T[] o = (T[]) **new** Object[count];  **for** (**int** i = 0; i < count; i++) {  o[i] = objs[i];  }  **return** o;  }  } |

2、泛型存储：使用对象集合完成如下功能

1.员工信息的查看

2.删除员工(要求使用泛型)

（10分）

|  |
| --- |
| public class Test {    public static void main(String[] args) {  List<String> list=new ArrayList<String>();  list.add("李二");  list.add("李三");  list.add("李四");  System.out.println(list);  list.remove(1);  System.out.println(list);  }  } |

3、如何编写一个泛型方法，让它能接受泛型参数并返回泛型类型?（10分）

|  |
| --- |
| 编写泛型方法并不困难，你需要用泛型类型来替代原始类型，比如使用T, E or K,V等被广泛认可的类型占位符。最简单的情况下，一个泛型方法可能会像这样：     public V put(K key, V value) {     return cahe.put(key,value);  } |

4、你可以把List<String>传递给一个接受List<Object>参数的方法吗？（10分）

List<Object> objectList;

List<String> stringList;

objectList = stringList; //compilation error incompatible types

5、编写一个泛型方法，让它能接受泛型参数并返回泛型类型

（10分）

|  |
| --- |
| public V put(K key, V value) {  return cache.put(key, value);  } |

6、编写自定义泛型类，并包含泛型属性，泛型方法，并创建另一个泛型类继承该泛型类

（10分）

|  |
| --- |
| **public** **class** Order<T> {  **private** String orderName;  **private** **int** orderId;  **private** T t;  List<T> list = **new** ArrayList<T>();  **public** **void** add() {  list.add(t);  }  **public** String getOrderName() {  **return** orderName;  }  **public** **void** setOrderName(String orderName) {  **this**.orderName = orderName;  }  **public** **int** getOrderId() {  **return** orderId;  }  **public** **void** setOrderId(**int** orderId) {  **this**.orderId = orderId;  }    **public** T getT() {  **return** t;  }  **public** **void** setT(T t) {  **this**.t = t;  }  @Override  **public** String toString() {  **return** "Order [orderName=" + orderName + ", orderId=" + orderId + ", t=" + t + ", list=" + list + "]";  }    **public** <E> E getE(E e) {  **return** e;  }    }  **class** SubOrder<T> **extends** Order<Integer>{  List<T> list1 = **new** ArrayList<T>();  } |

7、定义泛型方法，该方法可以把一个数组转换为一个List。并测试这个方法。

|  |
| --- |
| static public <T> List<T> fromArrayToList(T[] a) {      return Arrays.stream(a).collect(Collectors.toList());  }  @Test  public void givenArrayOfIntegers\_thanListOfStringReturnedOK() {      Integer[] intArray = {1, 2, 3, 4, 5};      List<Integer> list        = Generics.fromArrayToList(intArray);      assertThat(stringList, hasItems(1, 2, 3, 4, 5));  } |

8、定义一个泛型接口MyMap， 有两个泛型参数K, V， 分别表示键值类型。然后实现该接口。并测试这个实现类

|  |
| --- |
| public interface Mymap<K, V> {  public K getKey();  public V getValue();  }  public class MymapImpl<K, V> implements Mymap<K, V> {  private K key;  private V value;  public MymapImpl(K key, V value) {  this.key = key;  this.value = value;  }  public K getKey() { return key; }  public V getValue() { return value; }  }  //测试  Mymap<String, Integer> mp1= new MymapImpl<String, Integer>("Even", 8);  Mymap<String, String> mp2= new MymapImpl<String, String>("hello", "world");  Mymap<Integer, Integer> mp3= new MymapImpl<Integer, Integer>(888, 888); |

9、定义一个泛型工厂类GenericFactory， 通过方法createInstance(), 建立这个泛型类的对象。并测试。

|  |
| --- |
| public class GenericFactory<T> {  Class theClass = null;  public GenericFactory(Class theClass) {  this.theClass = theClass;  }  public T createInstance()  throws IllegalAccessException, InstantiationException {  return (T) this.theClass.newInstance();  }  }  //测试代码  GenericFactory<MyClass> factory =  new GenericFactory<MyClass>(MyClass.class);  MyClass myClassInstance = factory.createInstance(); |

10、定义一个泛型集合，实现Iterable接口。

|  |
| --- |
| public class MyCollection<E> implements Iterable<E>{  public Iterator<E> iterator() {  return new MyIterator<E>();  }  }  //MyIterator的定义  public class MyIterator <T> implements Iterator<T> {  public boolean hasNext() {    //implement...  }  public T next() {  //implement...;  }  public void remove() {  //implement... if supported.  }  }  //测试代码  public static void main(String[] args) {  MyCollection<String> stringCollection = new MyCollection<String>();  for(String string : stringCollection){  }  } |

11、实现一个自定义泛型类

（10分）

|  |
| --- |
| class Point<T>{ // 此处可以随便写标识符号，T是type的简称  private T var ; // var的类型由T指定，即：由外部指定  public T getVar(){ // 返回值的类型由外部决定  return var ;  }  public void setVar(T var){ // 设置的类型也由外部决定  this.var = var ;  }  }; public class GenericsDemo06{  public static void main(String args[]){  Point<String> p = new Point<String>() ; // 里面的var类型为String类型  p.setVar("it") ; // 设置字符串  System.out.println(p.getVar().length()) ; // 取得字符串的长度 }  }; |

12、定义泛型类，成员变量是Number类或其子类对象的数组，成员方法有：求数组元素最小值、最大值和平均值。创建泛型类对象，分别求Integer和Double型数组的最小，最大、平均值。

（10分）

|  |
| --- |
| package com.Shuzu3.java;  public class Array <T extends Number>{  public Array(T[] array)  {  this.array=array;  }  public void getMax()  {  //T[] array;  //double temp=(double) array[0];  //double temp=(double)array[0];  T max=this.array[0];  for(int i=1;i<this.array.length;i++)  {  if(array[i].toString().compareTo(max.toString())>0)  {  max=array[i];  }  }  System.out.println("数组元素中最大的值为: "+max);  }  public void getMin()  {  T min=this.array[0];  for(int i=1;i<this.array.length;i++)  {  if(array[i].toString().compareTo(min.toString())<0)  {  min=array[i];  }  }  System.out.println("数组元素中最小的值为: "+min);  }  public void getAver()  {  double ave=0.0;  double sum=0.0;  //sum=0;  for(int i=0;i<array.length;i++)  {  sum+=array[i].doubleValue();  }  ave=sum/array.length;  System.out.println("该数组的平均值为: "+ave);  }  private T[] array;  } |

13、使用数组来实现泛化的集合工具（10分）

|  |
| --- |
| **public** **class** MyArray<E> {  **private** **final** E[] arr;  @SuppressWarnings("unchecked")  **public** MyArray(**int** size) {  **if** (size<0) {  **throw** **new** IllegalArgumentException("Invalid Array Size:"+size);  }  arr = (E[]) **new** Object[size];  }    **public** **int** length() {  **return** arr.length;  }    **public** E get(**int** index) {  **return** arr[index];  }    **public** **void** set(**int** index,E e) {  arr[index] = e;  }    **public** **int** indexOf(E value) {  **for** (**int** i = 0; i < arr.length; i++) {  E element = arr[i];  **if**(element == value) {  **return** i;  }  **if** (element!=**null** && element.equals(value)) {  **return** i;  }  }  **return** -1;  }  } |