1.说一下泛型原理，并举例说明（源码）

https://blog.csdn.net/zx_android/article/details/79460755

Java 泛型的实现方法：类型擦除

Java 的泛型是伪泛型。在编译期间，所有的泛型信息都会被擦除掉。

类型擦出（type erasure）：

Java 中的泛型基本上都是在编译器这个层次来实现的。在生成的 Java 字节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，会在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，在编译后都会编程 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 的泛型实现方法与 C++模版机制实现方式之间的重要区别。

如：

```
ArrayList<String> arrayList1=new ArrayList<String>();
arrayList1.add("abc");
ArrayList<Integer> arrayList2=new ArrayList<Integer>();
arrayList2.add(123);
```

```
System.out.println(arrayList1.getClass()==arrayList2.getClass());
```

我们通过 arrayList1 对象和 arrayList2 对象的 getClass 方法获取它们的类的信息，最后发现结果为 true。说明泛型类型 String 和 Integer 都被擦除掉了，只剩下了原始类型。

如 2：

```
ArrayList<Integer> arrayList3=new ArrayList<Integer>();
arrayList3.add(1);//这样调用 add 方法只能存储整形，因为泛型类型的实例为 Integer
arrayList3.getClass().getMethod("add", Object.class).invoke(arrayList3, "asd");
```

在程序中定义了一个 ArrayList 泛型类型实例化为 Integer 的对象，如果直接调用 add 方法，那么只能存储整形的数据。不过当我们利用反射调用 add 方法的时候，却可以存储字符串。这说明了 Integer 泛型实例在编译之后被擦除了，只保留了原始类型。

2. 泛型中 extends 和 super 的区别

自己看总结
https://itimetraveler.github.io/2016/12/27/%E3%80%90Java%E3%80%91%E6%B3%9B%E5%9E%8B%E4%B8%AD%20extends%20%E5%92%8C%20super%20%E7%9A%84%E5%8C%BA%E5%88%AB%EF%BC%9F/

<? extends T>和<? super T>是 Java 泛型中的“通配符（Wildcards）”和“边界（Bounds）”的概念。

<? extends T>：是指 “上界通配符（Upper Bounds Wildcards）”

<? super T>：是指 “下界通配符（Lower Bounds Wildcards）”

1.什么是上界？

下面代码就是“上界通配符（Upper Bounds Wildcards）”：

Plate<? extends Fruit>

翻译成人话就是：一个能放水果以及一切是水果派生类的盘子。再直白点就是：啥水果都能放的盘子。这和我们人类的逻辑就比较接近了。

如果把Fruit和Apple的例子再扩展一下，食物分成水果和肉类，水果有苹果和香蕉，肉类有猪肉和牛肉，苹果还有两种青苹果和红苹果。

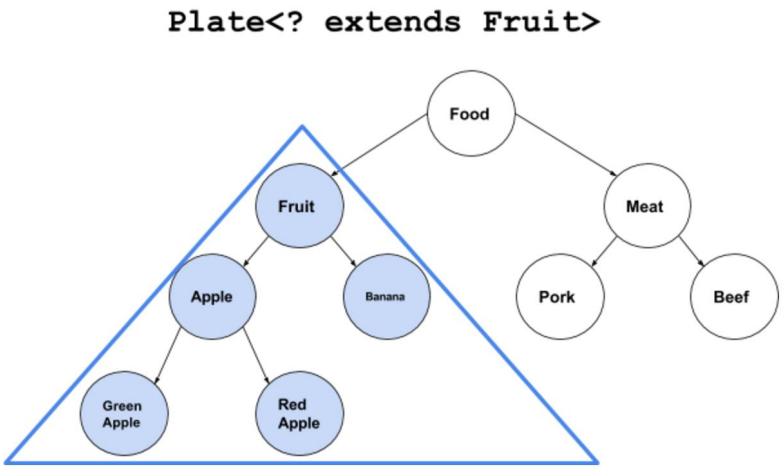
```
//Lev 1
class Food{}

//Lev 2
class Fruit extends Food{}
class Meat extends Food{}

//Lev 3
class Apple extends Fruit{}
class Banana extends Fruit{}
class Pork extends Meat{}
class Beef extends Meat{}

//Lev 4
class RedApple extends Apple{}
class GreenApple extends Apple{}
```

在这个体系中，下界通配符 Plate<? extends Fruit> 覆盖下图中蓝色的区域。

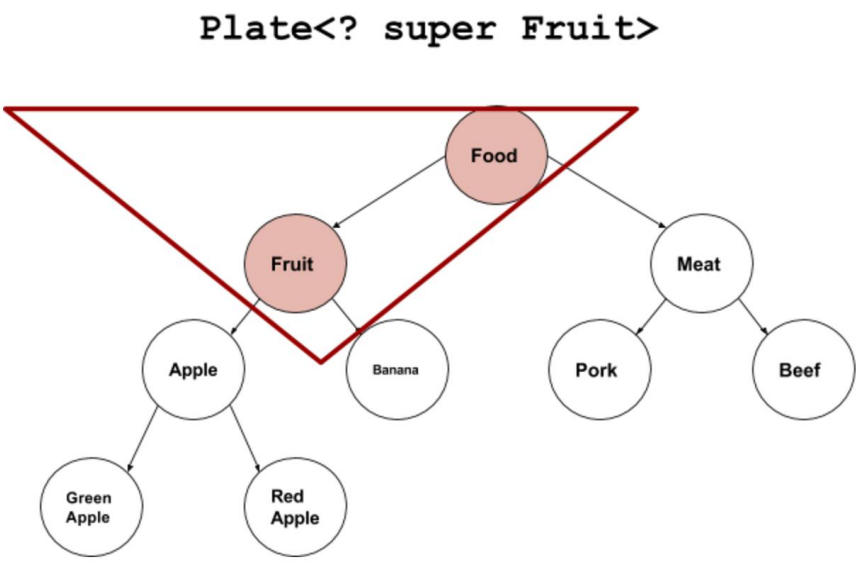


2.什么是下界？

相对应的，“下界通配符（Lower Bounds Wildcards）”：

Plate<? super Fruit>

表达的就是相反的概念：一个能放水果以及一切是水果基类的盘子。Plate<? super Fruit>是 Plate<Fruit>的基类，但不是 Plate<Apple>的基类。对应刚才那个例子，Plate<? super Fruit>覆盖下图中红色的区域。



不同之处：

（1）上界<? extends T>不能往里存，只能往外取

原因是编译器只知道容器内是 Fruit 或者它的派生类，但具体是什么类型不知道。可能是 Fruit？可能是 Apple？也可能是 Banana，RedApple，GreenApple？编译器在看到后面用 Plate 赋值以后，盘子里没有被标上有“苹果”。而是标上一个占位符：CAP#1，来表示捕获一个 Fruit 或 Fruit 的子类，具体是什么类不知道，代号 CAP#1。

然后无论是想往里插入 Apple 或者 Meat 或者 Fruit 编译器都不知道能不能和这个 CAP#1 匹配，所以就都不允许。

（2）下界<? super T>不影响往里存，但往外取只能放在 Object 对象里

使用下界<? super Fruit>会使从盘子里取东西的 get( )方法部分失效，只能存放到 Object 对象里。set( )方法正常。

因为下界规定了元素的最小粒度的下限，实际上是放松了容器元素的类型控制。既然元素是 Fruit 的基类，那往里存粒度比 Fruit 小的都可以。但往外读取元素就费劲了，只有所有类的基类 Object 对象才能装下。但这样的话，元素的类型信息就全部丢失。

总结:

PECS 原则:最后看一下什么是 PECS（Producer Extends Consumer Super）原则，已经很好理解了：

（1）频繁往外读取内容的，适合用上界 Extends。

（2）经常往里插入的，适合用下界 Super。