**<? extends T> Vs <? super T> - ProducerExtends ConsumerSuper**

**Basic object structure**   
class Animal {  
}  
  
class Carnivores extends Animal {  
}  
  
class Herbivores extends Animal {  
}  
  
class Tiger extends Carnivores {  
}  
  
class Lion extends Animal {  
}  
  
class Cow extends Herbivores {  
}  
  
class Zebra extends Herbivores {  
}

**Case- I – Define list as read-only, make it as producer**

**List<? extends Animal> animalList = new ArrayList<>(); 🡺 Producer**  
  
Animal animal = new Animal();  
Herbivores herbivores = new Herbivores();  
Carnivores carnivores = new Carnivores();  
Lion lion = new Lion();  
Tiger tiger = new Tiger();  
Cow cow = new Cow();  
**//The following lines will not be compiled, because it is a producer**  
**animalList.add(animal); // Compilation Issue  
animalList.add(herbivores); // Compilation Issue  
animalList.add(lion); // Compilation Issue  
animalList.add(tiger); // Compilation Issue  
animalList.add(cow); // Compilation Issue**

**Case- II – Add the data to the list and then make it read-only.**

List<Animal> list = new ArrayList<>();  
  
Animal animal = new Animal();  
Herbivores herbivores = new Herbivores();  
Carnivores carnivores = new Carnivores();  
Lion lion = new Lion();  
Tiger tiger = new Tiger();  
Cow cow = new Cow();  
  
list.add(animal);  
list.add(herbivores);  
list.add(lion);  
list.add(tiger);  
list.add(cow);  
  
List<? extends Animal> animalList = new ArrayList<>();  
**animalList = list;//Now it becomes read-only, You cannot add anything**

**animalList.add(new Zebra());//Compilation Issue  
animalList.add(herbivores);//Compilation Issue**

**what about new Animal(), it will also give compilation issue.**

**animalList.add(new Animal());//Compilation Issue**

**But you can remove from the readonly list;**

/\*\*  
 \* Here the list is the considered as readonly, there is no need to define

\* as unModifiable list.  
 \* This is the best approach because it provides  
 \* information at compile time but where as  
 \* Collections.unModifiableList() provides information  
 \* only at the Runtime  
 \*  
 \* **@param** list  
 \*/  
 public static void onlyTraverse(List<? extends Animal> list) {  
 for (Animal animal : list)  
 System.out.println(animal);  
  
**// list.add( new Zebra());//Compilation Issue as you cannot add** }

You can think why can’t I use Collections.unModifiableList(). Yes you can use it, but it will throw exceptions at runtime only.

In case of UnModifiable List, you can not add and remove, but in case of readonly list using ? extends T, you can delete.

List newList = Collections.*unmodifiableList*(list);  
*tryToTraverse*(newList);

public static void tryToTraverse(List<Animal> list) {  
 list.add( new Zebra() );//It will throw Exception only at runtime.  
}

**? extends T is producer, you can only iterate the list of elements, but you cannot add the elements, but you can remove.**

<http://stackoverflow.com/questions/2723397/java-generics-what-is-pecs>

Suppose you have a method that takes as its parameter a collection of things, but you want it to be more flexible than just accepting a Collection<Thing>.

**Case 1: You want to go through the collection and do things with each item.**  
Then the list is a **producer**, so you should use a Collection<? extends Thing>.

The reasoning is that a Collection<? extends Thing> could hold any subtype of Thing, and thus each element will behave as a Thing when you perform your operation. (You actually cannot add anything to a Collection<? extends Thing>, because you cannot know at runtime which specificsubtype of Thing the collection holds.)

**Case 2: You want to add things to the collection.**  
Then the list is a **consumer**, so you should use a Collection<? super Thing>.

The reasoning here is that unlike Collection<? extends Thing>, Collection<? super Thing> can always hold a Thing no matter what the actual parameterized type is. Here you don't care what is already in the list as long as it will allow a Thing to be added; this is what ? super Thingguarantees.

The principles behind this in Computer Science is named after

* Covariance - ? extends MyClass,
* Contravariance - ? super MyClass
* Invariance/non-Variance - MyClass

PECS (short for "**Producer extends and Consumer super**") can be explained by : **Get and Put Principle**

Get And Put Principle (From Java Generics and Collections)

It states,

1. use an **extends wildcard** when you only **get** values out of a structure
2. use a **super wildcard** when you only **put** values into a structure
3. and **don’t use a wildcard** when you **both get and put**.

**<? Super T> as consumer**

List<? super Animal> list = new ArrayList<>();  
list.add(animal);  
list.add(carnivores);  
list.add(herbivores);  
list.add(tiger);  
list.add(lion);  
list.add(cow);

The above works fine and there is no compilation issue.

List<? super Carnivores> carList = new ArrayList<>();  
carList.add(animal);//Compilation Issue  
carList.add(carnivores);  
carList.add(herbivores);//Compilation Issue  
carList.add(tiger);  
carList.add(lion);//Compilation Issue, Lion extends Animal  
carList.add(cow);//Compilation Issue

public static void consume(List<? super Animal> list ) {  
 **for(Animal animal : list) {** //Compilation Issue  
 **}** //But the following is fine  
 for (int i = 0; i < list.size(); i++) {  
 System.*out*.println(list.get(i));  
 }  
 **list.add(new Zebra());//Ok**}

**<? Super T> as Producer**

public static void asProducer(List<? extends Animal> list) {  
 //You can only iterate the list, you cannot add the elements  
 for(Animal animal : list)  
 System.*out*.println(animal);  
}

**Practical usage of PECS**

import java.util.ArrayList;  
import java.util.List;  
public class Util {  
  
 /\*\*  
 \* Iterate from the source list and  
 \* copy to the destination list  
 \*/  
 public static <T> void copy(List<? extends T> srcList, List<? super T> destnList) {  
 for(T t : srcList) {  
 destnList.add(t);  
 }  
 }  
  
 public static void main(String[] args) {  
 Animal animal = new Animal();  
 Carnivores carnivores = new Carnivores();  
 Herbivores herbivores = new Herbivores();  
 Tiger tiger = new Tiger();  
 Lion lion = new Lion();  
 Cow cow = new Cow();  
  
 List<Animal> srcList = new ArrayList<>();  
 srcList.add(animal);  
 srcList.add(carnivores);  
 srcList.add(herbivores);  
 srcList.add(tiger);  
 srcList.add(lion);  
 srcList.add(cow);  
 List<Animal> destnList = new ArrayList<>();  
 copy(srcList,destnList);  
 for(Animal an : destnList)  
 System.out.println("Animal Obj :::"+an);  
 }  
}

**PECS—Producer extends, Consumer super**

* **use Foo<? extends T> for a T producer**
* **use Foo<? super T> for a T consumer**
* **Only applies to input parameters**
* **Don’t use wildcard types as return types**

Complete Example is given below.  
  
public class TestPECS {  
  
 **public static <T> void copy(List<? extends T> srcList, List<? super T> destnList) {  
 for (T t : srcList)  
 destnList.add(t);  
 }**  
 public static void asProducer(List<? extends Animal> list) {  
 //You can only iterate the list, you cannot add the elements  
 for(Animal animal : list)  
 System.*out*.println(animal);  
 }  
  
 public static void asConsumer(List<? super Animal> list) {  
 //You can not iterate the list using advanced for loop  
 //You can add the elements to the list  
 Animal animal = new Animal();  
 Carnivores carnivores = new Carnivores();  
 Herbivores herbivores = new Herbivores();  
 Tiger tiger = new Tiger();  
 Lion lion = new Lion();  
 Cow cow = new Cow();  
  
 list.add(animal);  
 list.add(carnivores);  
 list.add(herbivores);  
 list.add(tiger);  
 list.add(lion);  
 list.add(cow);  
 }  
  
 public static void main(String[] args) {  
 Animal animal = new Animal();  
 List<Animal> list = new ArrayList<>();  
 *asProducer*(list);  
 *asConsumer*(list);  
 }  
}

List<? **extends** Product> readonlyList 🡺 Producer, Unmodifiable list or Readonly List

List<? **super** Product> list 🡺 Consumer, You can add objects

1. Use the <? extends T> wildcard if you need to retrieve object of type T from a collection.
2. Use the <? super T> wildcard if you need to put objects of type T in a collection.
3. If you need to satisfy both things, well, don’t use any wildcard. As simple as that.