**Get and Put Principle Producer Extends Consumer Super**

* **<? extends T> Producer Extends - use an extends to get values out of a structure**
* **<? super T> - Consumer Super - use a super to put values into a structure**
* **don’t use a wildcard when you both get and put.**

**List<? extends T>** 🡺 **Producer** 🡺**ReadOnly Data** 🡺 **Only for Traversing**/ **iterating 🡺 Can’t add, but remove**

* **Nothing can be added to the list**
* **list is readonly and visible at compile time, not required to define unModifiable list**
* **Collections.unModifiableList() provides information only at the Runtime**
* **But you can remove from the readonly list**
* **You can not iterate the list using advanced for loop**

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

**public static <T> void** produce1(**List<? extends T> list**) {  
 list.forEach( a -> System.***out***.println("Value: "+a));  
}  
  
**public static void** produce(**List<? extends String**> list) {  
 list.forEach(a -> System.***out***.println("Value: "+a));  
}

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

**List<? extends Animal> animalList = new ArrayList<>(); 🡺 Producer, you can only traverse the collection**  
  
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**

**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, nothing can be added**

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.

**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>.**

You actually cannot add anything to a Collection<? extends Thing>, because you cannot know at runtime which specific subtype 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 Thing guarantees.

The principles behind this in Computer Science is named after

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

**<? 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, Herbivores extends Animal  
carList.add(tiger);  
carList.add(lion);//Compilation Issue, Lion extends Animal  
carList.add(cow);//Compilation Issue, Cow extends Herbivores

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));  
 }

**public static void** consume(List<? **super** String> list, String value) {  
 list.add(value);  
}  
  
**public static** <T> **void** consume1(List<? **super** T> list, T t) {  
 list.add(t);  
}

or you can write like this.

for(Object animal : list) {  
  
 }  
  
 **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**

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) {  
 srcList.forEach(t -> 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);  
 }  
}

This List<Animal> srcList = **new** ArrayList<>();

Cannot be written like this

List<? Extends Animal> srcList = **new** ArrayList<>();

Or

List<? super Animal> srcList = **new** ArrayList<>();

public class TestPECS {  
  
 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);  
 }  
}

Covariant, Contravariant and Invariant

Covariant would mean that a List<String> is a subtype of List<Object>, contravariant that a List<Object> is a subtype of List<String> and invariant that neither is a subtype of the other, i.e. List<String> and List<Object> are inconvertible types.

**Covariant**: a **Cat[]** is an **Animal[]** , **Contravariant**: an Animal[] is a Cat[]

**Invariant**: an **Animal[]** is not a **Cat[]** and a Cat[] is not an Animal[]

**Covariance**: **Covariance is about producers.**

**Contravariance**: **Contravariance is about consumers**.

**Covariance = narrowing conversion. producers**

**Contravariance = widening conversion. consumers**

**Invariance (in this context) = not convertible.**

**Generics**

This is also possible for Generics:

List<String> aList...

List<? extends Object> **covariantList** = aList;

List<? super String> **contravariantList** = aList;

**Covariance is about producer - returns**

**class** Child **extends** Parent {  
 @Override  
 **String** produce() {  
 **return new** String(**"Child"**);  
 }  
}

**class** Parent {  
 **Object** produce() {  
 **return new** Object();  
 }  
}

**Sub.getSomething is covariant** because it returns a subclass of the return type of Super.getSomething (but fulfills the contract of Super.getSomething()).

Example is given below  
**public class** Test1 {  
 **public static void** main(String[] args) {  
 Parent[] parents = **new** Parent[3];  
 **for** (**int** i = 0; i < 3; i++)  
 parents[i] = **new** Child();  
 **for**( Parent p : parents )  
 System.***out***.println(p);//Prints Child@1b6d3586, Child@1b6d3586 etc  
 }  
}

**Contravariance is about consumer – consumes or accepts**

class Super {

void consume(**String parameter** )

}

class Sub extends Super {

void consume(**Object parameter** )

}

**Sub.consume() is contravariant** because it takes a parameter of a superclass of the parameter of

Super.doSomething (but, again, fullfills the contract of Super#doSomething)

Arrays Are Covariant but Generics are not

Java, an array of type T[] may contain elements of type T or any subtype of T. For instance:

Number[] numbers = new Number[3];

numbers[0] = newInteger(10);

numbers[1] = newDouble(3.14);

numbers[2] = newByte(0);

Object[] arr = new String[10];

Basically, an Object[] is a super type of String[], because Object is a super type of String. This is not true with generics. So, the following declaration is not valid, and won't compile:

List<Object> list = new ArrayList<String>(); // Will not compile.

**Get/Put Principle**

In summary, we **use covariance when we only intend to take generic values out of a structure**. We **use contravariance when we only intend to put generic values into a structure** and we use an invariant when we intend to do both.

**Overriding Method is covariant.**

**class** Parent {  
 Object get() {  
 **return new** Object();  
 }  
}  
  
**class** Child **extends** Parent {  
 @Override  
 String get() {  
 **return new** String(**"Child"**);  
 }  
}

**Variance** is about relationships between classes with different generics parameters.

**Covariance example**

**class** Lotus **extends** Flower {  
}

**class** Rose **extends** Flower {  
}

**public class** Flower {  
}

**interface** Garden<T **extends** Flower> {  
 T getFlower();  
}

**class** RoseGarden **implements** Garden<Rose> {  
 @Override  
 **public** Rose getFlower() {  
 **return new** Rose();  
 }  
}

**class** LotusGarden **implements** Garden<Lotus> {  
 @Override  
 **public** Lotus getFlower() {  
 **return new** Lotus();  
 }  
}

***//Use of Covariance*****public static void** check2() {  
 Garden<? **extends** Flower> garden = **new** RoseGarden();  
 Flower flower = garden.getFlower(); 🡺 Producer  
 System.***out***.println(flower);  
}

**Contravariance Example**

**interface** FlowerLover<T **extends** Flower> {  
 **void** use(T flower);  
}

**class** AnyFlowerLover **implements** FlowerLover<Rose> {  
 @Override  
 **public void** use(Rose flower) {  
 System.***out***.println(**"I like all flowers!"**);  
 }  
}

***//Use of Contravariance*****public static void** check1() {  
 FlowerLover<? **super** Flower> person = **new** AnyFlowerLover();  
 person.use(**new** Rose()); 🡺 Consumer  
}