Java Generics PECS – Producer Extends Consumer Super

By Lokesh Gupta | Filed Under: Generics

Yesterday, I was going through some java collection APIs and I found two methods primarily used for adding elements into a collection. They both were using generics syntax for taking method arguments. However, first method was using `<? super T>` where as second method was using `<? extends E>`. Why?

Let's look at the complete syntax of both methods first.

This method is responsible for adding all members of collection "c" into another collection where this method is called.

```
boolean addAll(Collection<? extends E> c);
```

This method is called for adding "elements" to collection "c".

```
public static <T> boolean addAll(Collection<? super T> c, T... elements);
```

Both seems to be doing simple thing, so why they both have different syntax. Many of us might wonder. In this post, I am trying to demystify the concept around it, which is primarily called **PECS** (*term first coined by Joshua Bloch in his book Effective Java*).

Why Generics Wildcards?

In my last post related to java generics, we learned that generics is used for **type safety and invariant** by nature. A usecase can be list of Integer i.e. `List<Integer>`. If you declare a list in java like `List<Integer>`, then java guarantees that it will detect and report you any attempt to insert any non-integer type into above list.

But many times, we face situations where we have to pass a sub-type or super-type of a class as argument in a method for specific purposes. In these cases, we have to use concepts like **covariance (narrowing a reference)** and **contra-variance (widening a reference)**.

Understanding <? extends T>

This is the first part of **PECS** i.e. **PE (Producer extends)**. To more relate it to real life terms, let's use an analogy of a basket of fruits (i.e. collection of fruits). When we pick a fruit from basket, then we want to be sure that we are taking out only fruit only and nothing else; so that we can write generic code like this:

Fruit get = fruits.get(0);

In above case, we need to declare the collection of fruits as `List<? extends Fruit>`. e.g.

```
class Fruit {
   @Override
   public String toString() {
      return "I am a Fruit !!";
   }
}

class Apple extends Fruit {
   @Override
   public String toString() {
      return "I am an Apple !!";
   }
}

public class GenericsExamples
{
   public static void main(String[] args)
   {
      //List of apples
      List<Apple> apples = new ArrayList<Apple>();
      apples.add(new Apple());

      //We can assign a list of apples to a basket of fruits;
      //because apple is subtype of fruit
      List<? extends Fruit> basket = apples;

      //Here we know that in basket there is nothing but fruit only
      for (Fruit fruit : basket)
      {
         System.out.println(fruit);
      }

      //basket.add(new Apple()); //Compile time error
      //basket.add(new Fruit()); //Compile time error
   }
}
```

Look at the for loop above. It ensures that whatever it comes out from basket is definitely going to be a fruit; so you iterate over it and simply cast it a Fruit. Now in last two lines, I tried to add an `Apple` and then a `Fruit` in basket, but compiler didn't allowed me. Why?

The reason is pretty simple, if we think about it; the `<? extends Fruit>` wildcard tells the compiler that we're dealing with a subtype of the type Fruit, but **we cannot know which fruit as there may be multiple subtypes**. Since there's no way to tell, and we need to guarantee type safety (invariance), you won't be allowed to put anything inside such a structure.

On the other hand, since we know that whichever type it might be, it will be a subtype of `Fruit`, we can get data out of the structure with the guarantee that it will be a `Fruit`.

In above example, we are taking elements out of collection "`List<? extends Fruit> basket`"; so here this basket is actually producing the elements i.e. fruits. In simple words, when you want to ONLY retrieve elements out of a collection, treat it as a producer and use "`? extends T>`" syntax. "**Producer extends**" now should make more sense to you.

Understanding <? super T>

Now look at above usecase in different way. Let's assume we are defining a method where we will only be adding different fruits inside this basket. Just like we saw the method in start of post "`addAll(Collection<? super T> c, T... elements)`". In such case, basket is used for storing the elements so it should be called **consumer of elements.**

Now look at the code example below:

```
class Fruit {
   @Override
   public String toString() {
      return "I am a Fruit !!";
   }
}

class Apple extends Fruit {
   @Override
   public String toString() {
      return "I am an Apple !!";
   }
}

class AsianApple extends Apple {
   @Override
   public String toString() {
      return "I am an AsianApple !!";
   }
}

public class GenericsExamples
{
   public static void main(String[] args)
   {
      //List of apples
      List<Apple> apples = new ArrayList<Apple>();
      apples.add(new Apple());

      //We can assign a list of apples to a basket of apples
      List<? super Apple> basket = apples;

      basket.add(new Apple());      //Successful
      basket.add(new AsianApple()); //Successful
      basket.add(new Fruit());      //Compile time error
   }
}
```

We are able to add apple and even Asian apple inside basket, but we are not able to add Fruit (super type of apple) to basket. Why?

Reason is that basket is a **reference to a List of something that is a supertype of Apple**. Again, **we cannot know which supertype it is**, but we know that Apple and any of its subtypes (which are subtype of Fruit) can be added to be without problem (*you can always add a subtype in collection of supertype*). So, now we can add any type of Apple inside basket.

What about getting data out of such a type? It turns out that you the only thing you can get out of it will be `Object` instances: since we cannot know which supertype it is, the compiler can only guarantee that it will be a reference to an `Object`, since `Object` is the supertype of any Java type.

In above example, we are putting elements inside collection "`List<? super Apple> basket`"; so here this basket is actually consuming the elements i.e. apples. In simple words, when you want to ONLY add elements inside a collection, treat it as a consumer and use "`? super T>`" syntax. Now, "**Consumer super**" also should make more sense to you.

Summary

Based on above reasoning and examples, let's summarize our learning in bullet points.

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 it is.
4. In short, remember the term **PECS. Producer extends Consumer super**. Really easy to remember.

That's all for simple yet complex concept in generics in java. Let me know of your thoughts via comments.

Happy Learning !!